From Live Sequence Charts to Implementation

A study of the LSC specification, the execution of behavioral requirements and exploring the possibilities to use an LSC model to generate Java code

By

Thomas Homme
Jon Erik Ramsland

Masters thesis
in
Information and Communication Technology

Grimstad, May 2003

Summary

In the article ”From Play-In Scenarios to Code: An Achievable Dream” [8] D.Harel outlines an exiting new way of developing software with the aid of the new and expressively powerful language of live sequence charts (LSC) [1] for describing message sequencing.

The language of LSC is a highly visual approach for modeling behavioral requirements and is based on message sequence charts [13]. LSC introduces the ability to specify mandatory behavior in charts through the notion of “liveness”, events that must happen.

The development process described involves using a friendly capture method called “play-in” to create the behavioral requirements in the formal LSC language. Algorithms are then used to synthesize system behavior parts from the requirements and the system can in turn be verified against the requirements. Cyclic synthesis and verification is used until a sufficient system model has been created. From the system model, code and final implementation can be generated.

This thesis is a study of the language of LSC and how code generation can be applied in different stages of this development process. We present the constructs and elements of the LSC language and further extensions of the specification. The method for executing LSCs called “play-out” is also presented.

Three applications of code generation in the development process are introduced. We explore the possibility synthesizing system behavior in the form of statecharts from LSCs, and from there using developed tools to generate the actual code. Only an outline of a synthesis algorithm is available, and we present the main aspects of it. The “play-out” methodology is a way of executing LSCs. We outline a methodology for integrating a “play-out” engine into a partial implemented system in order to execute it as it was fully operational. This allows us to test the behavioral requirements or utilize the result as a final implementation. We also present a way of generating code directly from an LSC specification with certain constraints, and thus circumventing the need for a system model with behavior.

We have developed a tool called “LSC Visualizer”, which is able to read LSCs stored in our own XML format. We have also developed a prototype “play-out” engine that can be coupled with existing system models as well as a prototype for generating code directly from an LSC specification. An algorithm for building an event-tree that describes the partial order has been constructed, and is utilized by the “LSC Visualizer” and both prototypes.

Preface

This thesis was written for Agder University College as a part of the Norwegian "Master of Technology" title. The work has been carried out in the period between January 2003 and May 2003. Most of our work has been done in Kristiansand.

We would like to thank our supervisor Jan P. Nytun at Agder University College for valuable help and encouragement throughout the project period.

Grimstad, May 2003

Thomas Homme and Jon Erik Ramsland

Table of Contents

List of Figures

1 Introduction

1.1 Background

Live Sequence Charts (LSCs) are a relatively new technology. They are an extension of Message Sequence Charts (MSC), where one can specify the "liveness" of charts.

The "liveness" is the specification of what must happen in a chart. It also allows the possibility to distinguish between possible and mandatory system behavior. LSCs allow one to describe what must occur, what may occur and what should not occur.

LSCs have later been extended with new features, such as timing constructs, assignments, loops, variables and symbolic object instances. These features transfer LSCs from simply being charts to specify behavioral requirements to become an executable model. The ability to execute behavioral requirements allows one to test the system prior to implementation. Errors found and corrected in the early development phase of a project will be severely cheaper and easier than correcting errors after the system has been implemented.

Going from behavioral requirements like an LSC model to code or system model can speed up the development process; it might also yield more error free implementation, as there are no intermediate development stages where errors can occur.

1.2 Live Sequence Charts an Extension of Message Sequence Charts

Live sequence charts are an extension of message sequence charts and addresses weaknesses found in message sequence charts by introducing "liveness", events that must happen. LSC also allows specifying anti-scenarios, a safety property that forces the program to terminate when it enters a bad state.

LSC models make use of two kinds of charts, universal and existential. Universal charts are described as "restrictions over all executions of the system". The universal charts are also associated with a "prechart", which specifies the scenario, that when executed and recognized forces the system to satisfy the universal chart.

Existential charts are sample interaction of the system and must be satisfied by at least one possible execution of the system. This is what mostly resembles classical message sequence charts.

Conditions have been extended to have the possibility to specify possible and mandatory behavior. This is described as the temperature of the condition, and can be

either “hot” or “cold”. If a condition is “hot” it must be true or the system will terminate. A “cold” condition can be false and the system will gracefully exit subcharts or the current universal chart. This allows for the creation of branching constructs. A “hot” condition set to false in a universal chart will define a so called anti-scenario.

Messages have also been extended with temperature. A “hot” message must be received or the system will have performed an illegal operation and thus terminate. A “cold” message sent but not received will not constitute illegal behavior.

The original LSC specification has been extended with additional features that allows for execution of LSC models.

1.3 Execution of Live Sequence Charts

Execution of live sequence charts, often referred to as “play-out”, causes the executing application to react according to the universal parts of the live sequence chart specification, whereas the existential charts can be monitored to see if the execution of them was successfully completed.

The process of executing live sequence charts aims to equal the processes of running the final implemented application with the user interactions and environmental effects that would normally be there to affect the system. This allows for thoroughly examination of how the system works and detection of what may go wrong and what does actually happen, without implementing the final system.

The execution of live sequence charts should be a strictly rules based system which blindly follows and complies with all the preset rules. The execution of the system should never violate any rules described by the charts in effect whatsoever and neither should the system execute anything that is not explicitly stated.

Allowing developers to execute the behavioral requirements for testing a system could be a cost efficient method of developing systems. One of the reasons is that it requires a lot less work to redefine the behavioral requirements for the system model rather than reworking parts or the whole of a full implementation.

1.4 Code Generation from Live Sequence Charts

The method for executing live sequence charts called “play-out” aims to equal execution of the final implementation. For some reactive systems the “played out” behavior might be sufficient in the way that it captures the intended behavior. This would for some systems constitute using execution of behavioral requirements as the final implementation. A skeletal implementation of some parts of the system would exists, like class structure, window creation, database access functions and other parts of a

system that are not practically to describe with LSCs. Code would be generated into this "skeleton" to couple it with the LSC specification of the system. An implemented "play-out" engine would have to accompany such a solution.

There exist proposed methodologies for synthesizing statecharts from LSC requirements. If the statecharts produced are good enough, that is, they are able to express the synopsis of the LSCs without losing precision, other tools can be used to generate code from them. The code generated with this method might be of more value than the first approach since no "third-party" code, like a "play-engine", would have to be adapted. The generated statecharts would yield the system behavior and must be accompanied by a class structure in order to generate code.

To generate useable code directly from an LSC specification, without a "play-out" engine driving the execution or the need to create statecharts as an intermediate step would be very challenging. The LSC language is not designed for this, but simplifications of the language might be adapted to achieve this on a chart by chart basis.

1.5 Thesis Definition

<<The thesis will consist of a study of the LSC specification, its extensions and the execution of behavioral requirements described by LSCs. The thesis will also look into the possibilities concerning automatic code generation from LSCs. The thesis will present how LSCs differ from other forms of sequence charts, like the ITU specified MSC(Message Sequence Charts) or UMLs Sequence Diagrams, and how LSC can be useful throughout the development process, for describing behavioral requirements and validating the system model.

A methodology for executing behavior described by LSC, called "play-out", has been proposed. The students will present how this methodology works.

The thesis will also consist of a study of the possibility of automatic code generation from LSCs, and what value generated code has in the development process. The students will look into if it is feasible to generate Java code from behavioral requirements such as LSCs, and what would be the best approach of doing so. An extension or simplification of the LSCs might be proposed better to do so.

To generate code from an LSC model or use an LSC model to execute behavioral requirements the LSC model must be represented in a way a computerized tool can understand, one possibility is to use XML. The LSC model may have to be coupled with the system model, for this XMI could be used.>>

When studying the material for this thesis we saw the possibility the methodology for executing behavioral requirements called "play-out" could have in the development process. It could be deployed as a tool for testing an implemented system against the behavioral requirements, or more interestingly, to allow a "skeletal system" to be used as a final implementation. It could also be used as means of testing a not fully constructed system during development. In order to achieve this, code generation would have to be used. This would not so much be code generation from LSC, but rather code generation for LSC. We have, after discussion with our supervisor, therefore changed the thesis definition to encompass this with the following paragraph:

<<In addition to a study of how code can be generated from an LSC model the students will also examine the possibility of generating code in order to use the "play-out" methodology with an existing Java system model described by XMI.>>

1.5.1 Thesis Limitations

<<The code that will eventually be generated from XML/XMI should be limited to simple LSCs as the more complex ones would move the focus away from the usability of LSC and towards code generation in general. Also since this is a thesis more or less split in three aspects of LSCs the depth on each of those parts we can go into would be somewhat limited.>>

1.5.2 Security

<<Some of the algorithms concerning execution by "play-out" of LSCs in David Harel's papers are patent pending.>>

1.6 Our Work

Most of our work is based on papers publicly available on the Internet. Some of the tools described and used in these papers are not available to the public and neither to us. It was therefore up to us to develop tools that could simulate parts of these tools to the extent that it would satisfy the needs we had.

The main questions to be answered in this thesis are:

- What does live sequence charts offer for modeling a system that other kinds of sequence charts do not?
- How does the described methodology, "play-out", for execution of live sequence charts work?
- Is it feasible to generate code from live sequence charts, or in other ways generate code in such a way as to allow for the execution of live sequence charts?

To answer these questions we first look at what new functionality live sequence charts offers and which possibilities it opens up for with respect to modeling a system. We also look at how the execution of live sequence charts are done and which benefits it gives in regard to testing and discovering errors in the model before it is being implemented.

We have constructed a rudimentary XML format for describing simple LSC charts. These simple LSC charts include no constructs for symbolic messages, support for time enriched LSC or environmental effects. This format allows for a somewhat limited coupling with existing class skeletons. An algorithm for building a tree that describes the partial order of events within a chart was also developed.

Three different approaches for code generation have been studied and discussed. One possibility is to use already developed methods of synthesizing state based object systems from the LSC specification and use other tools do generate code from statecharts, thus yielding an implementation of the LSC model. Here we focus on presenting and evaluating the synthesis methodology. Another approach we have been studying is to use a "play-out" engine to drive the execution of an already developed simple system model. This can be used to test both the system model and the requirements, and also be used as a final implementation. The third possibility we have looked at is to create the code directly from the charts. To achieve this, the LSC language has to be simplified. Prototypes for the two latter approaches have been developed.

We also develop a prototype application that can read live sequence charts stored in our own XML format. This "LSC Visualizer" can draw charts described by the XML and it can also draw an event tree where it is fairly easy to follow the flow and order of messages.

1.7 Report Outline

In the following chapters we will take a closer look at some of the technologies mentioned above. In chapter 2 we will give an introduction to live sequence charts, the constraints within the language, its usefulness and present its advantages. A description of how the execution of live sequences charts works will be presented in chapter 3. We have constructed an XML format for LSC which will be described and explained in chapter 4. In chapter 5 we will present an algorithm for creating an event tree based on the events within a chart.

Automatic code generation and the possibility to do this with live sequence charts will be discussed in several different chapters. Chapter 6 will give an introduction to a development process where code generation could be applied at different stages and for different means. We look at code generation by synthesizing statecharts in chapter 7. In chapter 8 we present a methodology for incorporating a "play-out" engine into an partially implemented system. We also present the result of a prototype that utilizes this

approach. We take another approach in chapter 9, where several restrictions on the LSC language is proposed to allow for the possibility of generating code directly from a behavioral specification. A prototype for this approach and the results of it is also presented.

As a final part of this thesis report chapter 10 will contain a discussion of the technologies used and how well they worked out. And finally, chapter 11 will contain our own subjective conclusion of this thesis.

2 Live Sequence Charts

2.1 Introduction

Message sequence charts, specified by the International Telecommunication Union (ITU) [14], and the sequence diagrams incorporated into the Unified Modeling Language (UML) [15] have been used for the modeling of systems for several years. Live sequence charts (LSC) [1] has been introduced as an extension to these possibilities of modeling systems. Both LSC and sequence charts utilize the notion of partial order to restrict the order events within a chart, but there are significant weaknesses in the expressive power of the sequence charts, which LSC addresses.

One of the most significant weaknesses of sequence charts is that these do not have the possibility to distinguish between possible and mandatory behavior in scenarios. This change with LSCs, where it is possible to specify what behavior must be satisfied by all runs of a system, by introducing "liveness", events that must occur.

With the introduction of "liveness" to the charts it is now possible to separate possible and mandatory behavior in a system. LSC is also semantically powerful enough to describe behavior the system must not exhibit, called anti-scenarios.

The language of LSC has been extended with constructs for loops [2], assignments [2], symbolic variables [5], symbolic object instances [5] and timing [3]. [2] also extends the LSC specification with a non-deterministic choice construct and the notion of forbidden elements. These extensions enhance the LSC language, and alter it from being a descriptive language for the documentation and specification of behavioral requirements to becoming a language for describing complete executable models.

In this chapter we will present the different elements and extensions that make up the LSC language. We will also introduce concepts that are useful for reading and interpreting live sequence charts.

2.2 LSC Example

We have created LSCs for a Television application. The charts making up the behavioral requirements of the application can be found in its entirety in Appendix H. We will use extracts of these charts, where applicable, as examples for elements and constructs of the LSC language described below.

The Television system consist of instances for a GUI, called TV, that controls the system, a power control that keeps track of the power-state, a volume and a channel. A user operates the GUI which in turn sends messages to the other instances of the system.

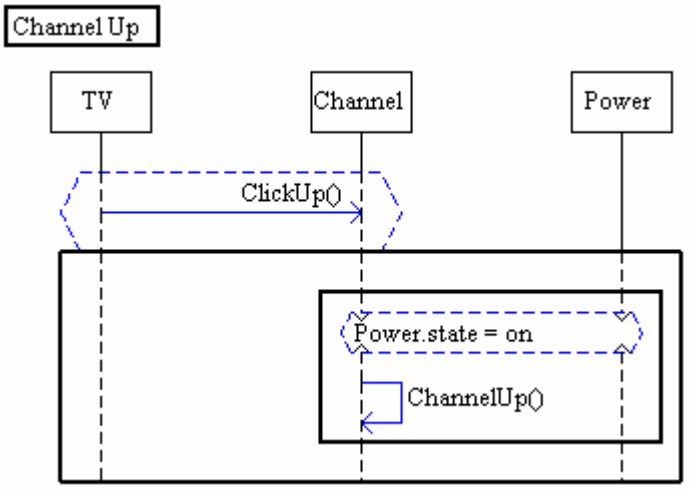

Figure 2.1: Changing the channel

Figure 2.1 shows a simple LSC for changing the channel upward on the TV. This chart describes that whenever the message "ClickUp()" is sent from the TV instance to the Channel instance, the system will check if the power is on, and if it is on, the channel will be changed. The various elements in this chart will be described in detail in the next section.

2.3 Elements and constructs of the LSC Language

In this chapter we will present the elements and constructs that make up the visual notation of LSCs as presented in [1] and extended in [2], [3] and [5]. Examples will be given from the Television application LSCs and other charts constructed for illustrative purposes.

2.3.1 Charts

There are two kinds of charts, universal charts and existential charts. Universal charts are defined as (from [1]) "restrictions over all possible system runs", while existential charts are used to specify sample system behavior.

Universal charts can be associated with a prechart or an activation message. A prechart specifies a scenario that if executed will force the system to satisfy the behavior described in the universal chart. An activation message is a single message that forces the system to satisfy the universal chart. The prechart and activation message can be seen as an IF statement with the body of the universal chart as the THEN statement.

Existential charts are used to describe possible behavior the system can exhibit, and should be satisfied by at least one run of the system.

Universal charts are drawn with solid border which denotes mandatory behavior, while existential charts are drawn with a dashed borderline which denotes possible behavior. This concept that solid lines are mandatory behavior and dashed lines are possible behavior is used for all LSC elements. Precharts and activation messages are also drawn with dashed lines.

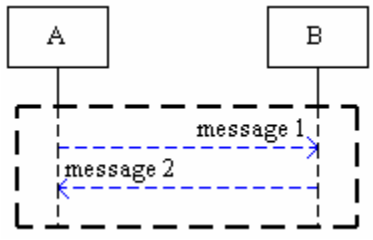

Figure 2.2: Existential chart

2.3.2 Messages

There are two kinds of messages; synchronous and asynchronous. A synchronous message has a solid arrowhead, while an asynchronous message has an open ended arrowhead. In addition a message has a temperature that can be either "hot" or "cold". If the temperature is "hot", denoted by a solid line, the message is mandatory; it must be sent and received. A "cold" message is drawn with a dashed line, and reception of the message is not mandatory. Figure 2.3 shows an example of the different kind of messages. The messages to the left are asynchronous, with the "hot" message above the "cold" one. "Hot" and "cold" synchronous messages are on the right side.

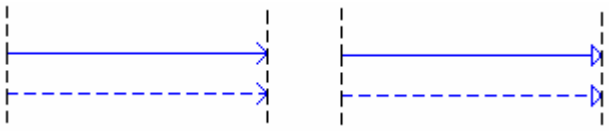

Figure 2.3: Different types of messages

2.3.3 Conditions

Conditions may be seen as branching constructs, and contains an expression to be evaluated. If an expression of a "hot" condition is evaluated to be false, the run must terminate. On the other hand, if the condition was "cold", the current subchart is exited, or the whole chart if it is on the top level. "Hot" conditions can be used to create anti-scenarios. This is done by describing the illegal scenario in a prechart with a "hot" condition in the main chart that will always be false.

Together with subcharts and loops, conditions can be used to create IF-THEN-ELSE, DO-WHILE and WHILE constructs. A condition is synchronized for one or more instance lines; this is denoted by small arrowheads pointing into the condition on entry and exit points for the synchronized instances. This means that the condition shall not be evaluated until all the instances involved reaches the synchronized locations, and neither might progress until the expression is evaluated.

Figure 2.4: "Cold" and "hot" conditions

2.3.4 Subcharts

Hierarchy in a chart is created with subcharts. Subcharts are LSCs and are specified for a set of the instances from the parent chart, but it might also contain new instances. Loops are created by using an iteration symbol in the top left corner of the subchart. Limited iteration is created by using a constant or a numeric value, while unlimited iteration is denoted by an asterisk.

A "cold" condition evaluated to be false will exit a subchart. A condition in the top of an iterated subchart will create a WHILE construct, while a condition in the bottom of the chart will create a DO-WHILE construct. Figure 2.5 shows two kinds of loops. On the left there is a constant loop, in this scenario the message "ping()" will be sent four times. The subchart on the right shows a DO-WHILE construct, the message "ping()" will be sent until the variable response in the instance Receiver becomes true.

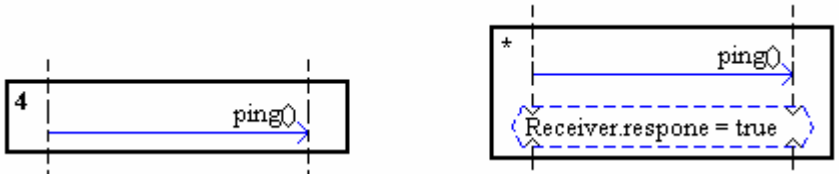

Figure 2.5: Loops

An IF-THEN construct is created with a subchart with a "cold" condition at the beginning denoting the IF statement. If it is true, the subchart will be executed, if it is false, the subchart will exit. IF-THEN-ELSE constructs are created with two subcharts with the IF condition in the top of the first chart, if it is evaluated as false, the first subchart will exit and the second will be executed, otherwise, the first chart will be executed and when it is done execution will omit the second subchart. In Figure 2.6 both IF-THEN and IF-THEN-ELSE are shown. In the subchart to the left the message "ChannelDown()" will be sent if the variable state in the instance Power is true. The

subchart to the right behaves similar, but if the variable state is not true, the message "PowerOn()" will be sent.

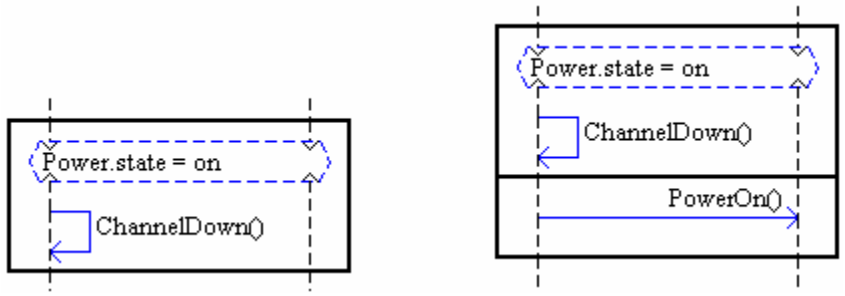

Figure 2.6: IF-THEN and IF-THEN-ELSE

2.3.5 Assignments

[2] proposes an assignments extension for LSC. Assignments allow temporary storage of the values of properties of objects or the result of functions on such values. The validity of an assignment is for the given chart only. As with conditions, assignments can be synchronized for one or more instances, thus forcing the instances not to progress until the assignment is reached and executed. Figure 2.7 shows how the variable state from the instance Power is stored in the local chart variable X.

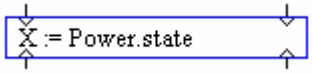

Figure 2.7: Assignments

2.3.6 Instances and locations

An instance is drawn as a rectangle around the name of the object that is instantiated. The instance line is the vertical line that goes from the instance rectangle through the chart. Locations are the points along the instance line where instance line and events intersect. Such an event can be the sending or reception of messages, evaluation of conditions and assignments or the entry or exit of a subchart. A location may have several message events, but only one condition, assignment or subchart entry or exit event.

A location is also associated with a temperature, which indicates how the instance should progress beyond the location along the vertical instance line. If the location is "hot" the execution of the chart must progress beyond the location, but if the temperature is "cold" progress is only provisional, and may occur. The location at the end of the chart, called the "maximal location", must be "cold". The instance line will be drawn in dashed style from a "cold" location and in solid style from a "hot" location.

Figure 2.8 shows an example of "hot" and "cold" locations. After "message 1" is sent and received progress along the instance lines is no longer mandatory, and it might be possible that "message 2" will not be sent without violating the chart.

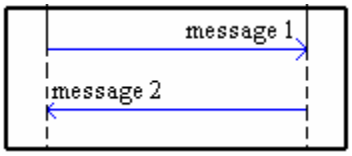

Figure 2.8: "Hot" and "cold" locations

2.3.7 Symbolic Instances

An ordinary instance only reference one object, while a symbolic instance can reference different objects that instantiates the same class. A single chart can thereby compactly describe scenarios that otherwise would require multiple charts.

The notation of a symbolic instance is the class name followed by "::", and represents all the objects of the class in the system. There are two different possibilities for binding a specific instance of this set to the chart.

The first is when a symbolic instance is used in the charts prechart. Instead of just monitoring a single instance, all the objects that instantiates the class that the symbolic instance refers to are monitored. When one of the objects actually performs the prechart scenario it is bound and used in the chart.

If the symbolic instance is not used in the prechart it must be bound in the main chart, this is done by using an expression to select the set (there can be more than one) of instances to be bound. The notation of this parameterization of a symbolic instance is a "thought-bubble" from the instance.

The binding can be either "hot" or "cold", denoted by a solid or dashed "thought-bubble" respectively. The effect of the temperature is noticeable only when the expression yields a set of more than one instance. If the binding is "cold", one instance from the set is selected at random and used in the chart. In a "hot" binding the whole set is chosen, and a new copy of the chart is created for each of them.

A punctuation before a variable in the binding expression is considered a self-reference to that variable in the class the symbolic instance refers to. If for instance we have a class with a variable ID of the type Integer, and we want the instance in our chart to apply for all instances with an ID greater than ten, we simply write ".ID > 10" in the expression bubble. The binding would have to be "hot" in order apply to all the instances, and not just a randomly selected instance with ID greater than ten.

Local variables for a given chart, from assignments, can also be used in the symbolic instance expression and thereby greatly extending the expressive power of symbolic instances. If the binding uses such variables, the binding should be performed right after the assignment is performed.

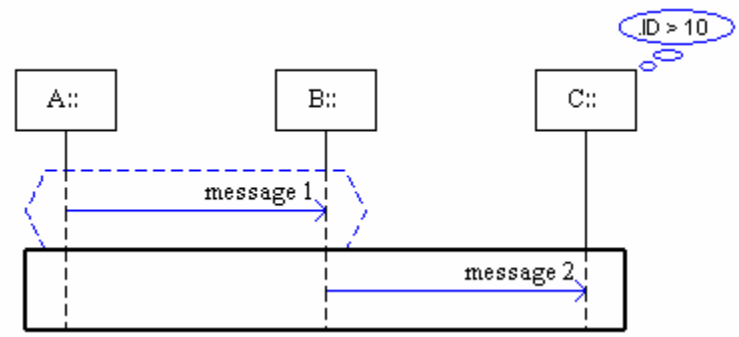

Figure 2.9: Symbolic instances

Figure 2.9 shows an example of both types of symbolic instance binding. All objects that instantiates the classes A and B are monitored. The object that sends "message 1" and the object that receives the message are bound when such a message occurs. The object that is bound for the class B in the prechart sends "message 2" to all instances of the class C with the value of the "ID" attribute greater than ten.

2.3.8 Symbolic Messages

To enhance the expressive power of messages, the definition of a message was extended to handle parameters [5]. Each message can have zero or more parameter variables of a given type. With the help of these variables further generalization of charts is possible. A single chart can for instance be used describing what happens when a switch is turned off or on, instead of two different chart describing two separate scenarios. With the introduction of symbolic messages, the partial order of events was also extended (see 2.4). Figure 2.10 shows the use of symbolic messages when turning on or off the Television.

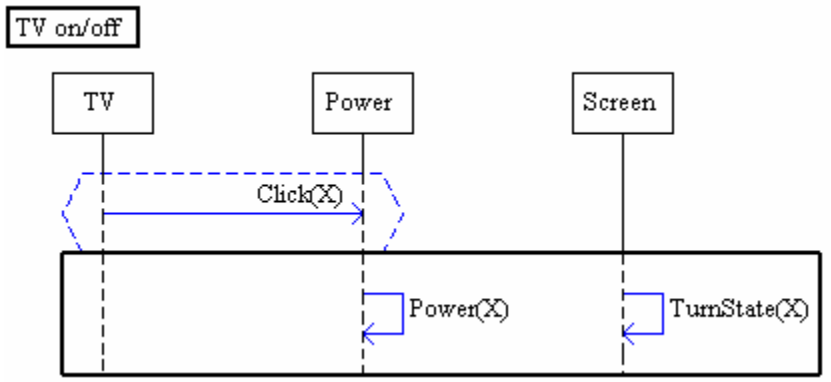

Figure 2.10: Turning on or off the Television

2.3.9 Forbidden elements

A chart is violated once a message appears when it is not expected. Sometimes it is convenient to be able to say a certain message should not appear although it is not related to the currently active chart, or that a given condition must not hold when a certain chart is active. These invariants can be expressed with forbidden elements proposed in [2]. Forbidden elements are associated with a specified constrained scope, for instance a subchart or an entire chart. These elements are denoted as being "hot" or "cold". A "cold" forbidden element that is violated the current constrained scope exits gracefully. If a "hot" forbidden condition is evaluated to be true while in the constrained scope it is considered to be a violation of the requirements.

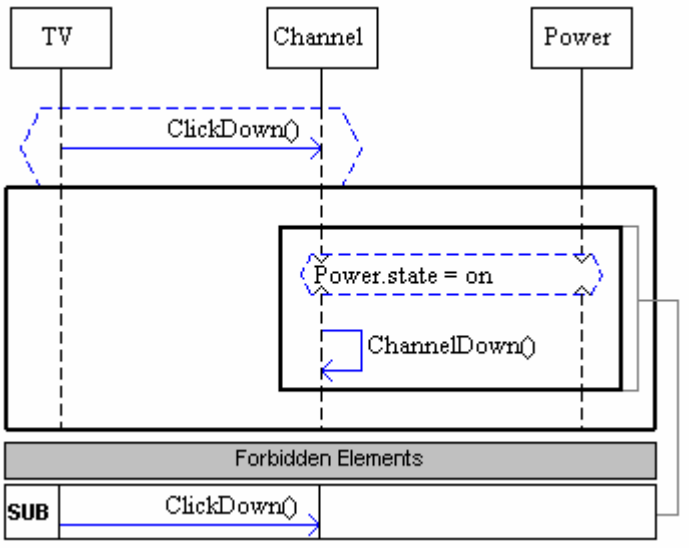

Figure 2.11: Forbidden elements

Figure 2.11 shows an example of a chart with forbidden elements defined. In the chart the message "ClickDown()" is not allowed to occur when the subchart is active.

2.3.10 Time enriched LSC

[3] extends the LSC specification with a special clock object with one message, "Tick". This clock is represented by a special instance in a LSC, drawn as a clock symbol. The clock object has a time property that conditions and assignments may gain access to. When this is done a special hourglass symbol is drawn in the top right corner of the element. Assignments can be used to store the time at one point and conditions can be used to specify minimal and maximal delay.

A minimal delay is written as "Time > Stored-Time + Min-Delay". A "hot" condition containing an expression on this form will be continuously evaluated until enough time passes for it to become true. If the condition containing the minimal delay is "cold", the chart will exit if the minimal delay is evaluated to be false, and no waiting will be forced. An example of a minimal delay is shown in Figure 2.12, after the message "poll()" is sent at least five ticks of the clock will occur before the message "respond()" is sent.

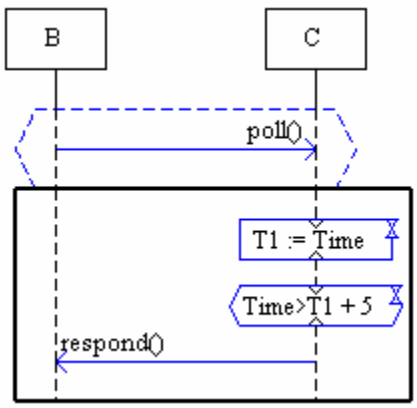

Figure 2.12: A minimal delay

A maximal delay is written as "Time < Stored-Time + Max-Delay". A "hot" condition describing a maximal delay is evaluated and if found to be false the executing will abort. Since time cannot go backwards, once a maximal delay expression becomes false, it will always be false. A "cold" maximal delay evaluated to be false will exit the current chart or subchart.

Maximal and minimal delays may be used together with other conditions. When this is done, they should be separated by a semicolon.

The special Tick message of the clock instance can be used to specify time events. By using the Clock symbol in a chart and placing the Tick message in the prechart, one can

in the main chart specify the scenarios that should occur at every tick of the clock. Figure 2.13 shows an example of this. Every time a "Tick" message occur a "cold" condition is evaluated. The time modulo 60 is calculated and the result is compared to zero. This condition will be true once every minute and the main chart will be executed, forcing the message "poll()" to be sent from instance B to instance C.

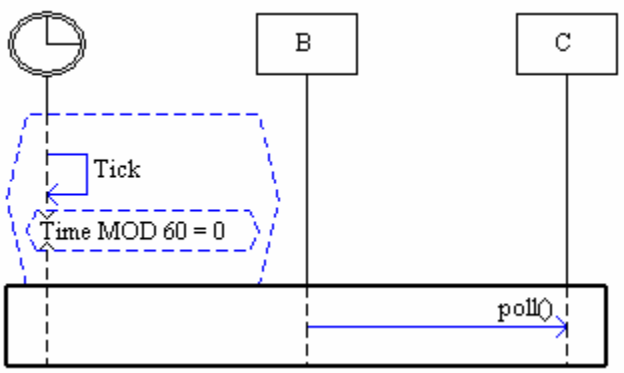

Figure 2.13: Time constraints and events

2.3.11 Non-deterministic choice

A SELECT command is placed inside a condition. The command has a parameter P that represents the true to false ratio. If for instance the command "SELECT(50, 50)" is placed in the condition of a IF-THEN-ELSE construct, the THEN part will be executed half the time and the ELSE part the other half of the time.

2.3.12 External objects

The language of live sequence charts is extended to be able to react to external events. This can include, but is not limited to sensors, computers and the nature. Such elements are referred to as the systems environment.

2.4 Partial order of events

The partial order of events ($<_m$) induced by the LSC m, describes the order of which the events in the chart should occur. An event can be the sending of a message, the reception of a message, the evaluation of a condition, to perform an assignment, the entry of a subchart or prechart or the exit of a subchart or prechart. The intersection between an event and an instance line is a location. A location can have several events, but limited to only one condition, assignment or entry and exit of subchart or prechart.

The location of an instance is written in < > brackets with the instance name followed by the location number separated by a comma. Location numbers can be seen in Figure 2.14

along the instance line. These numbers are for illustration purposes only and should not be a part of a chart.

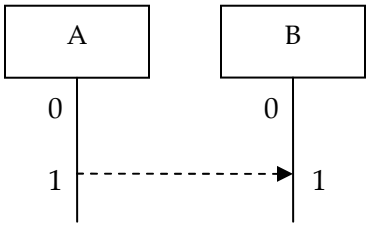

Figure 2.14: Message from A to B

Events along the instance lines are ordered from top to bottom. An event higher up on the instances line happens before one lower down ($\langle A,0\rangle <_m \langle A,1\rangle$). Further order is induced by the sending of a message between instances ($\langle A,1\rangle <_m \langle B,1\rangle$) and by the transitive closure of these ($\langle A,0\rangle <_m \langle B,1\rangle$).

With synchronous messages, the sender is blocked until the message is received. This causes the reception of the message to occur before any other event that follows sending the message in the instance sending the message ($\langle B,1\rangle <_m \langle A,2\rangle$), which is not the case for asynchronous messages.

Other elements that describe the ordering of events are synchronization blocks, like conditions and assignments. None of the instances that are involved in the synchronization may progress until the element involved in the synchronization has been processed, and the element may not be processed before all instances reach the synchronization block. Events in any of the instances involved must therefore occur before events in any of the instances after the synchronization block.

Subcharts work in much the same way as synchronization blocks. All instances participating in the subchart are synchronized before entering the subchart and before exiting it. Precharts are a variant of subchart and work in the same fashion.

The symbolic variables that [5] introduces also extends the partial order. The vertical position of events containing the same symbolic variable imposes that ordering upon them. This is only specified for the first occurrence of a variable, and forces the first occurrence of the variable to come before all other occurrences.

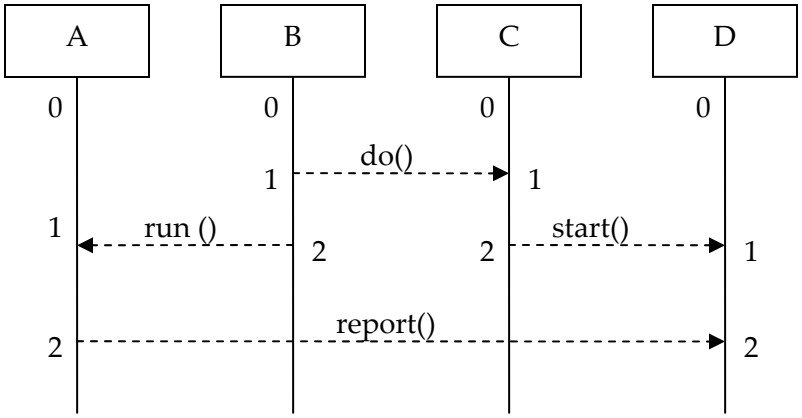

Figure 2.15: Basic chart with locations marked by numbers

Figure 2.15 shows a simple chart with four instances sending the messages "do()", "run()", "start()" and "report()" between each other. As the arrowhead indicates these messages are synchronous. If we look at the first instance (A), we see that its first event is the reception of "run()" and the next the sending of "report()". From instance B we see that "do()" is sent and later "run()" is sent. As the messages are synchronous we can for simplification consider the sending and receiving of each message as a single event.

From the two instance A and B we can draw the conclusion that the messages occur in this order: "do()", "run()", "report()". From instance C we see that "do()" is received and then "start()" is sent and for instance D "start()" is received and then "report()" is received. For the two instances C and D we conclude that the order of the messages is "do()", "start()", "report()".

If we look at the four instances we observe that "do()" must come first, but there is no specific order of the next two messages; "run()" and "start()". During a run of this chart, the order of "run()" and "start()" is therefore irrelevant, but both must occur before "report()". Figure 2.16 shows a graph describing the order of events. Chapter 5 describes how to build such a tree from the events gathered from the instances of a tree.

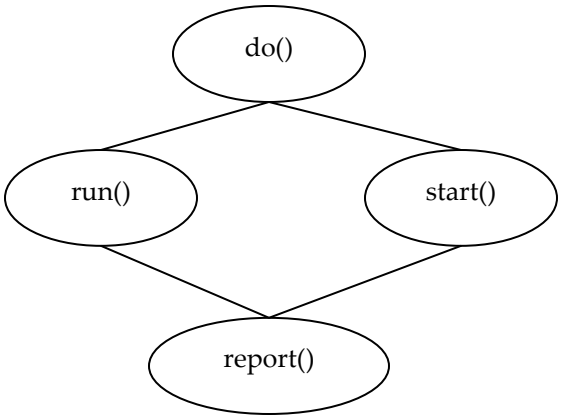

Figure 2.16: Partial order of events tree

2.5 Cut

A cut of a chart describes the progress of each instance along the vertical instance line and should be consistent with the partial order of events. This implies that a valid cut should not indicate progress that would violate the partial order. A cut can also be seen as a state of the chart.

The notation of a cut is written as the sequence of locations that makes up the progress of a given chart. The set of locations for all the instances in the chart, separated by commas, makes up the cut. A cut is typically placed in parenthesis. The instance name can be dropped for simplicity, a cut would then be the location numbers listed for each instance from left to right separated by commas in a parenthesis.

From Figure 2.15 a valid cut could be (<A,1>,<B,2>,<C,1>,<D,0>) ("do()" has been sent and received, as has "run()"), while (<A,2>,<B,2>,<C,2>,<D,1>) would be an invalid cut. Here "run()", "do()" and "start()" would have been sent and received, but the message "record()" would only have been sent and not received, as the message is synchronous, the cut is illegal.

There are a finite number of valid cuts for a single chart, for the chart in Figure 2.15 these would be: {(0, 0, 0, 0), (0, 1, 1, 0), (1, 2, 1, 0), (0, 1, 2, 1), (1, 2, 2, 1), (2, 2, 2, 2)}

2.6 Run

A run is the set of cuts that make up a successful transition of a chart, starting with the cut (0, 0, 0, 0). For a given chart there can be many different runs.

The chart in Figure 2.15 has two different legal runs, these are: {(0, 0, 0, 0), (0, 1, 1, 0),(1, 2, 1, 0), (1, 2, 2, 1), (2, 2, 2, 2)} and {(0, 0, 0, 0), (0, 1, 1, 0),(0, 1, 2, 1), (1, 2, 2, 1), (2, 2, 2, 2)}

2.7 Violating a Chart

Events that are not visible in a chart are not restricted by the chart. Therefore if an event is not defined explicitly in a chart, it is allowed to occur in between the other events in the chart without violating it. If we consider a scenario where the first message "Click()" has occurred in the prechart in Figure 2.17. The prechart would then only be traced successfully if the second message event "Click()" from instance B to instance C occur, and only be violated if the first "Click()" message from instance A to instance B occurs again.

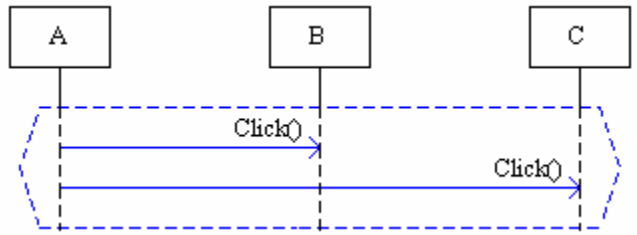

Figure 2.17: A prechart

Sets of restricted messages can be defined explicitly for charts. This can be useful for defining restrictions for messages that does not appear in the chart.

2.8 Summary

The elements and constructs presented in this chapter show a significant extension of the classical message sequence charts. Using the notion of an elements temperature is a powerful to describe "liveness" in charts. This yields a language for the construction of more descriptive behavioral requirement models. The extensions make it possible to start to serious look at synthesis of system model behavior from charts and execution of such charts.

In the following chapter we will see how a methodology for executing live sequence charts work.

3 Executing LSC by "play-out"

3.1 Introduction

"Play-Out" is a proposed methodology in [2] for executing behavior described by live sequence charts. The following chapter will discuss how this technology works and what constraints that needs to be taken into consideration. There have already been implemented a tool known as "play-engine" for recording and playing out LSCs, although it is not available to the public at the time of writing. Some of the "play-engines" possibilities will be presented at the end of this chapter. In most cases user interaction must be taken into consideration because these will affect how the system operates.

3.2 User Interaction

Systems developed require at times user interaction for them to operate in the intended way. It could for instance be a system as simple as a TV, the basic functionality is already there, implemented. However the TV does not act on its own; it is completely dependant on a user to operate it, for example changing the volume level or channels, turning it on and off and so on. Therefore, the "play-out" mechanism of live sequence charts may require user interaction for the charts to be processed correctly, it being live interaction or predefined routines.

3.3 Playing Out Behavior

Play-Out is a methodology for executing live sequence charts. Playing out a scenario is a process of testing the system behavior for a system that has been described by live sequence charts. When a system is played-out the system is tested with user actions and how it responds to the interaction with the user and environment.

Using a system in a "play-out" scenario by the end users does not require knowledge about live sequence charts, use cases or requirements specified for the system. All the end users need to know is how to operate the system as it had already been implemented as the final system.

When executing a system it will usually not behave the exact same way each time it is being executed, mostly because they are dependent on user interaction which would cause different behavior, unless it is a system only meant to operate in one specific way. The underlying aspect of the execution of live sequence charts should be a strictly rules based system which blindly follows and complies with all the preset rules no matter what. The execution of the system should never violate any rules described by the charts in effect whatsoever and neither should the system execute anything that is not explicitly stated.

The "play-out" of a system has two different ways of being executed. These two different possibilities are "step" and "super-step". When in "step" mode, the user is responsible for the execution of the system in the way that he has to manually select when the system should continue, while in "super-step" mode the system only requires user interaction when the system would normal require this in the every day use of the system, thus "super-step" will execute all consecutive operations where no user interaction is required.

When the system is being played out the different charts are activated when they are needed. For instance when a user powers on a TV the chart that controls the powering on function is activated and executed, and when the chart has finished it is deactivated.

The state that the execution is in on a given time has been described as a "cut" in section 2.5. Whenever a specific functionality of the system is used the relevant chart should track the behavior of the system to know at any given time what conditions and evaluations that has been completed, and what messages has been sent and received.

3.4 Play-Out Scenarios for Testing of a System

When a system is being planned for implementation it is very important that extensive tests are carried out to debug the model in order to prevent redesign of the system in a later phase which would require a lot more work to be redone than if the bugs are detected in the early phases of development. From a cost efficient viewpoint this is very desirable. This raises a desire for executable models.

The language of live sequence charts includes both what are referred to as universal charts and existential charts. Universal charts can be seen as restrictions that should be satisfied by all system runs, while existential charts describes behavior that the system can exhibit, and that should be satisfied by at least one system run. The universal charts are serving to drive the "play-out" of the systems and describe how the charts interact with each other and react to events by other charts. The existential charts on the other hand may be used for testing and evaluation of the system and should be monitored in the "play-out" methodology.

It would be natural to allow for these test runs to be stored in some way, XML has been proposed in [2] for this purpose. This would allow for the same specific runs to be executed several times and allows for a study of what happens during execution of a specific run as the model changes. When doing it this way the user would also have the option of altering the specification of the run and execute them again to see how the changes would affect the system behavior.

3.5 The execution mechanism

The execution mechanism proposed in [2] is used for playing out live sequence charts. The following subchapters describe the architecture for this and which main components that are included in this mechanism. When charts are being executed they will enter and exit different states as they are activated and deactivated. These states are explained below.

3.5.1 States of the charts

As the lifecycle of universal and existential charts are different, the states will be different as well and it would therefore be reasonable to separate which states apply to each of them.

3.5.1.1 States applying to universal charts

Not existing
There exists no copy of the original chart at this point. When a copy is needed it will be created.

PreActive mode
When one of the minimal events of a prechart occurs or one of the minimal conditions is met the chart will enter preactive mode, that is, the state where it will evaluate the precharts.

Active mode
Active mode is the state a chart will enter when a prechart has been successfully evaluated. This is where the main activity and events of most normal charts will be performed

3.5.1.2 States applying to existential charts

Not existing
There exists no copy of the original chart at this point. When a copy is needed it will be created.

Monitored mode
An existential chart where one of the minimal events happens or one of the minimal conditions is met; the chart will be transferred into monitored mode. The chart will here be monitored for all activity.

3.5.2 The copy of an LSC

Live sequence charts are used to specify behavioral requirements of a system, where the relevant charts are activated when the prechart is successfully completed. Each and every one of the charts may become active several times during a system run. Whenever one of these charts is activated the system does not work on the chart directly, whereas it should make a copy of the original chart and execute the copy instead. This is done to be able to identify the charts uniquely when the same chart is active multiple times during a system run.

The live sequence charts and their copies during a specified system run have certain constraints that need to be satisfied for the system to be played-out. As several copies of a chart might be active during a run or there might be several active at a given time, there need to be a way to identify these charts, this is done as described below.

With a given live sequence chart L the live copy (denoted by C_L) is defined as following: **C_L = < L, M, Cut>** Where L denotes the given copy of the original chart including distinct copies of the original charts variables. M is an element that can be either in PreActive, Active or Monitored mode during execution. Cut is a legal cut of the chart representing the current location of the execution of the instances of L in this particular copy.

An event in the chart L described above, is a minimal event if there is no event e′ in L such that $e' <_L e$, where $<_L$ denotes the partial order that is induced by the live sequence chart L. Partial order has been described in section 2.4.

An event e can be enabled with respect to a cut C if the location of every instance that is participating in the event is at the location exactly prior it. There can be no event $e' <_L e$ that has not been processed. There is usually only one participating instance, but some constructs may have several participating instances.

The last definition for an event is that an event e violates a chart in the given cut C if the event appears in the chart but has not been enabled with respect to the cut.

3.5.3 Chart life cycle

The chart life cycle are different for universal and existential charts and will thus be explained separately. What is common for both of these charts are that a copy do not exist initially until a specific event occur and as a result a copy of the relevant chart will be created and brought into the appropriate mode.

3.5.3.1 Life cycle of a universal chart

The copy of the original charts life cycle is defined as following: If the copy of the chart does not exists and a minimal event or one of the minimal conditions of the prechart occurs in a universal chart a copy is created and transferred to preactive mode.

A chart in preactive mode may have two different outcomes. The prechart may finish normally, which will cause the chart life cycle to enter active mode. If the prechart is violated by a condition which evaluates to false or a message that should not occur, the chart will be exited and the copy deleted.

A chart which reaches active mode has several different outcomes depending on the conditions and the temperature on the cut. If the cut is "hot" when the chart is violated by an event, that is sending or receiving of a message, the chart will always abort because an illegal run was produced. The chart will abort if a "hot" condition is evaluated to false as this will produce an illegal run as well. When a "cold" condition in a chart is evaluated to false the chart will exit and be deleted. If this "cold" condition is evaluated to false in a subchart it will only exit the subchart and which in turn will only change the order of events and does not cause the copy to terminate. When a chart is violated by a "cold" event it will exit and be deleted. If none of the above occurs, the chart will finish and then be deleted.

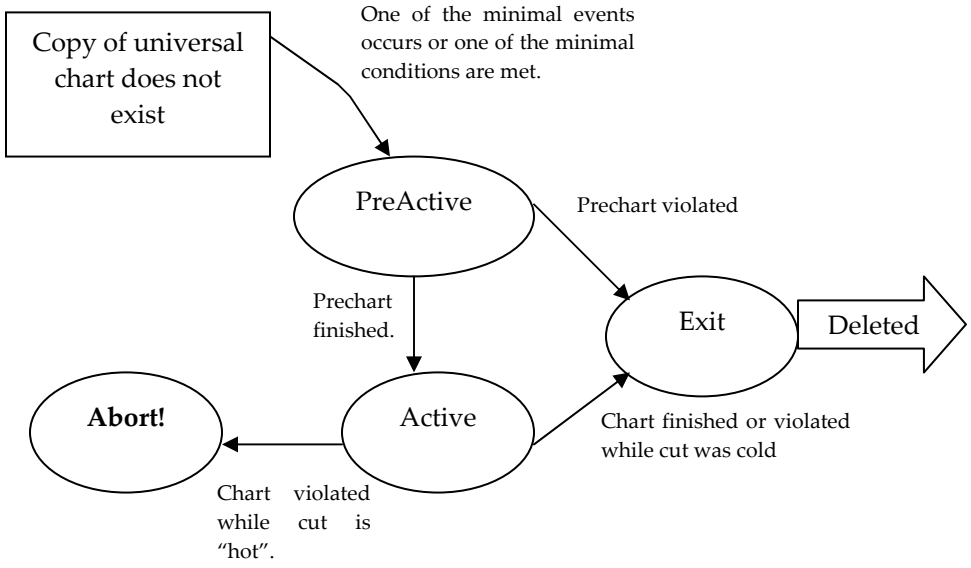

Figure 3.1: Life cycle of a universal chart

3.5.3.2 Life cycle of an existential chart

If one of the minimal events of the existential chart occurs, or one of its minimal conditions is met, the chart will enter monitored mode.

A monitored chart that is traced successfully will enter the "completed state" before being deleted. If the chart is violated while traced it will be deleted immediately.

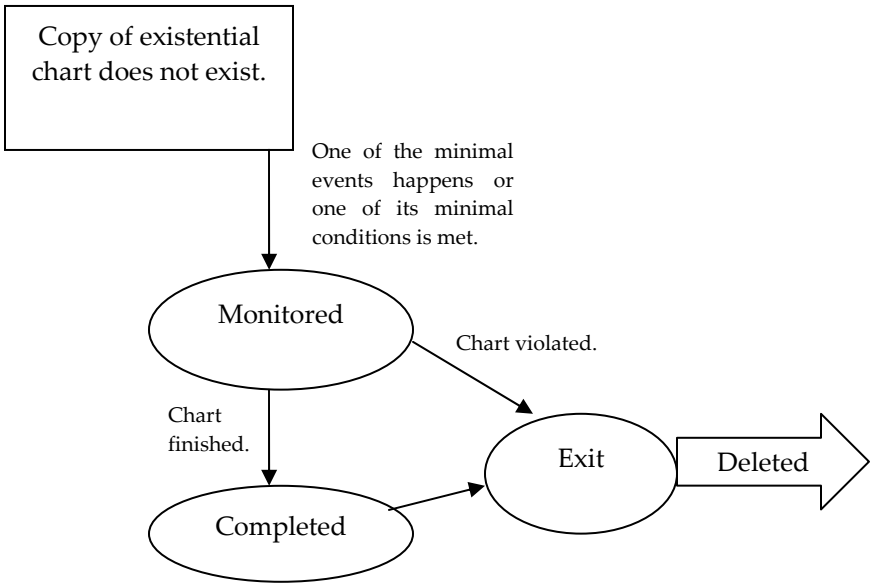

Figure 3.2: Life cycle of an existential chart

3.6 Play-out example

In this paragraph an example of "play-out" of one of the live sequence charts from the TV application will be presented. This chart example describes what happens when the "VolumeUp" button is clicked. TV represents the user operating the system.

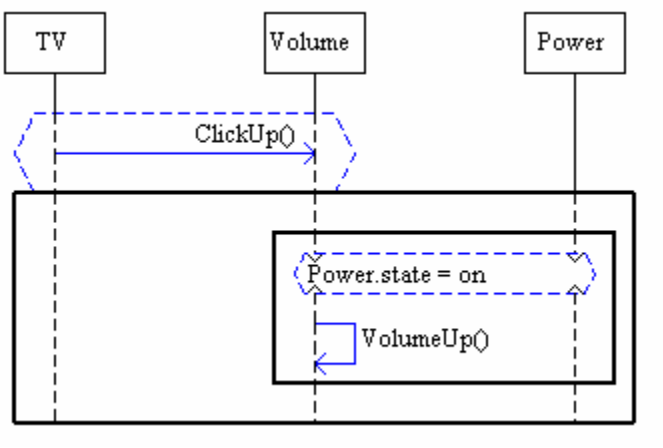

Figure 3.3: Initial state of the chart

Figure 3.3 shows the initial chart, as it has been described by LSC. At this point there is no copy of the chart, but, the system will create a copy, C_L, of the chart once one of the minimal events occurs or conditions are met. In this case the minimal event is the message "ClikUp()" from TV to Volume. When this message occurs and is identified for this chart a copy of the chart is made and the chart is transferred to preactive mode. In preactive mode all events in the prechart will be monitored and conditions will be evaluated. If the evaluation is successful the chart will transfer to active mode, if unsuccessful the chart will exit and C_L will be deleted. In this case there are no more events in the prechart, and the chart will enter active mode.

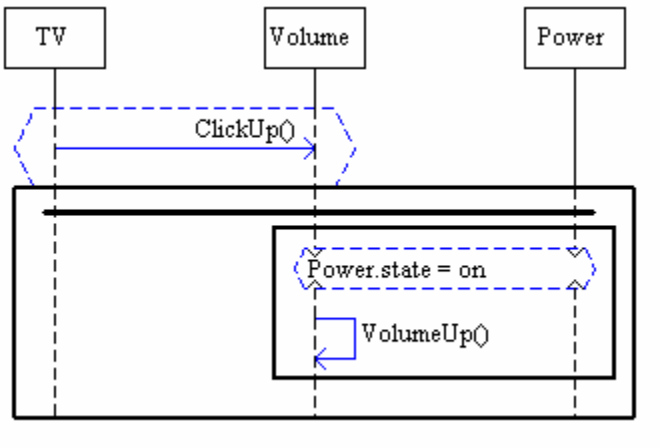

Figure 3.4: Cut of the chart in active mode

Figure 3.4 shows the chart in active mode, with the current cut drawn as a black solid line across the vertical instances lines. The universal main chart has been entered and subcharts, conditions and other events will be enabled and performed in order to drive the execution. Figure 3.5 shows the two different paths the execution of the chart can

take. The first thing that will happen is that the subchart will be entered, and the condition will be enabled since all instances in its synchronization block are at the location exactly prior to it. Next the condition will be evaluated. If it is evaluated as false, the subchart will exit, but if it is true, the execution will flow as normal, and the message event "VolumeUp()" will be enabled. The enabled message event will be executed, and the exit of subchart event will be enabled.

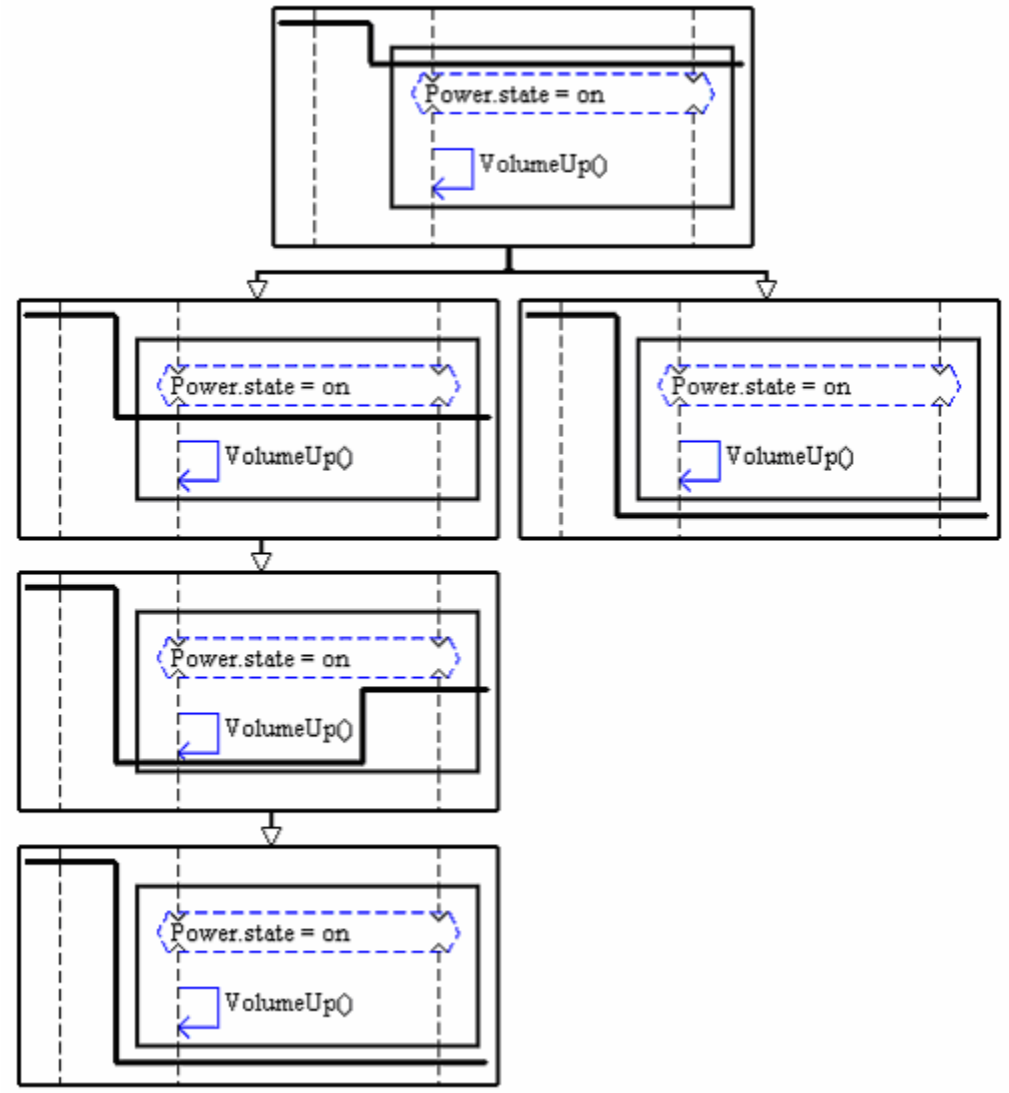

Figure 3.5: Two paths the execution can take

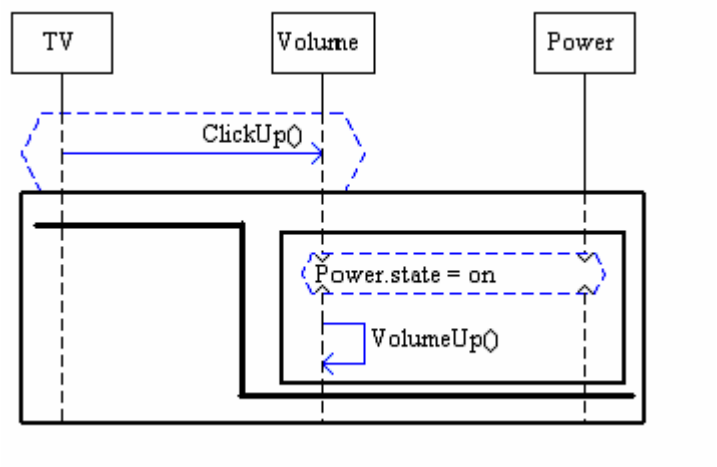

Figure 3.6: Maximal locations reached

The chart in Figure 3.6 shows a cut of the chart after the subchart has been executed. There are no more events to be sent or conditions to be evaluated and thus all the instances in the chart will reach their maximal location and the copy of C_L will be deleted. Execution of the chart has been successful.

3.7 Smart "play-out"

Smart "play-out" is a proposed method in [7] of executing behavioral requirements described by live sequence charts with aid of formal analysis methods. The model checking done by these formal analysis methods is a way of detecting a correct "super-step" or to prove that no such "super-step" exist. A "super-step" is defined as a method executing all consecutive events that does not require any user input to the system. If the "play-out" task of the charts is defined as a verification problem a produced counter-example will constitute the desired "super-step". This will however only work on universal charts as the existential charts can not be limited to one single "super-step", since the existential charts can contain external events that alone can trigger "super-steps". However, doing the same for existential charts does not require a lot of modification of the process. When doing smart "play-out" for existential charts, cooperation of the environment is assumed.

Smart "play-out" can be said to be driven by these counter-examples that have been produced by the formal analysis. The partial order within the language of live sequence charts or the many interleaving charts may cause "bad" nondeterministic choices. The very concept of counter-examples makes it possible to avoid this.

The general approach of smart "play-out" is to formulate the execution of the charts as a verification problem and use a counterexample provided by model-checking as the

desired super-step. The system this is to be used on must be constructed according to the universal charts of the specification.

The transition of the model from a "play-out" problem to a model-checking problem is done because the semantics for live sequence charts are only defined for one chart in itself when testing reactive systems. The objective of the translation is to find a correct behavior when several charts are acting together. Parts of this include having one process for each of the actual objects represented by the charts. The translation methods for the different elements and constructs are described in [7].

3.8 Playing in Behavior – The "play-engine" environment

The "play-engine" as mentioned previously is a tool described in [2] for developing systems. The basic operation of the "play-engine" is to develop a GUI in for example Visual Basic, and then connect it with the "play-engine". You would then operate the GUI as it would be done in a final implemented application and the "play-engine" would record these operations and develop live sequence charts based on this. These recorded runs can then be stored in XML. The "play-engine" also allows for describing LSC directly and thus the need for "play-in" is not mandatory.

The "play-engine" can import and execute the stored charts, which are referred to as "play-out". The GUI would then be operated as normal and the engine itself takes responsibility that events are not violated.

The "play-in" of such a scenario causes the "play-engine" to create the needed live sequence charts based on all the actions of the user when being played in. The user will for example push a button; turn a knob or something along the lines of this. Then the action for the button pressed, knob turned or whatever will be defined as system behavior for each run this action occurs.

The "play-in" part of the engine allows the developer and to some extent the user to apply function based values on variables. This allows the developer to assign methods for solving external activities that cannot easily be applied to the program at a later stage. This feature allows the "play-engine" environment to be set down with a way for solving for example a complicated mathematical function or evaluation a complex algorithm or retrieve data from a database.

3.9 Summary

"Play-out" as a methodology has been proposed as a way of executing behavioral requirements. Copies of the charts are made and activated as they are needed and the copy is executed. There are two main states for universal charts, preactive when the prechart is being traced and active when the main chart is being executed. Universal

charts are defined as restrictions over all runs, while existential charts needs to be satisfied by at least one possible run.

"Play-out" being a strictly rules based concept, should execute charts and avoid doing anything that will cause the system to violate either of the charts. This means having to cross check between all activate charts that the next action to be taken will not violate anything in either of the other charts. It should not execute anything not stated in the charts either.

The "play-out" methodology has a proposed extension in "Smart Play-Out" as formal analysis methods for the execution of the live sequence charts. It uses counter-examples as means of verification testing and it is also possible with this method to avoid "bad" non-deterministic choice, a choice that can lead to the violation of a chart in the future.

In the next chapter we will present an XML format we have constructed for describing charts with the most fundamental LSC elements.

4 XML for LSC

4.1 Introduction

For LSCs to be processed by a computerized tool they must be described in a way that is easy for software to understand. An XMI [17] format for describing LSCs are under development by Omega [22], but was not available during our work on this thesis. We therefore had to construct our own format. XML [16] is an excellent choice for this purpose as it is easy to parse while still being easy to edit manually.

We have constructed a rudimentary XML format for describing simple LSC charts. It is roughly based on the way sequence diagrams (UML) are represented in XMI. Our XML format is limited in the way that there are no constructs for symbolic instances, time enriched LSCs or communication with the environment. Furthermore there can only be one event per location, and a subchart can only reference the same instances as the main chart. Some of these limitations are imposed deliberately for easier manual editing.

In this chapter we will present the different model elements, and how they are constructed in the XML format. We will also present a methodology for creating XML for a given chart and to bind a system model described by XMI to a LSC model in XML. An XML Schema for the format can be found in Appendix B.

4.2 The Language

Each model element has a unique id attribute that can be used to reference it from other elements. Valid model elements are <model>, <chart>, <instance>, <message>, <prechart>, <subchart>, <condition> and <assignment>.

A chart is associated with a name, which is gathered from the contents of a <name> tag placed in the body of the <chart> tag. Charts can contain instances, messages, precharts, subcharts, conditions and assignments. A chart may be either existential or universal; which is determined by the type attribute in the <chart> tag. Several charts making up a complete LSC specification can be placed between <model> tags. Figure 4.1 shows a sample of the different tags without references.

```
<chart id="" type="universal">
    <instance…>
    <message…>
    <condition…>
    <assignment…>
    <prechart…>
    <subchart…>
</chart>
```

Figure 4.1: Chart

Instances are described by <instance> tags. The name of the instance is written between <name> tags, and placed in the <instance> tags body.

For each instance we specify its locations from top to bottom. Each location may only have one event. This is not the case in the specification, where a location can have several message events, but we limited it to one event. Valid events are prechart entry, prechart exit, main chart entry, message sending or receiving, assignment and condition. Each event in the listing is only a reference to the actual event, this is done to make the location listing more lucid and to avoid multiple definition of the same event. Whether a message event is sending or reception is not taken into account in the listing. The message is only referenced and who is the sender and who is the receiver can be seen in the actual message.

```
<instance id="">
<name></name>
        <locations>
           <prechart event="entry" idref=""/>
           <message idref=""/>
           <prechart event="exit" idref=""/>
           <chart idref=""/>
       <assignment idref=""/>
       <subchart event="entry" idref="A"/>
       <condition idref=""/>
       <subchart event="transition" from="A" to="B"/>
       <subchart event="exit" idref="B"/>
       <message idref=""/>
        </locations>
</instance>
```

Figure 4.2: Instance

Figure 4.2 shows quite an extensive list of different events that make up the locations of an instance. Events that are actually in the precharts should be placed between the prechart entry and exit events; the same applies to subcharts. The figure also shows the connection of two subcharts, this is typically done when constructing IF-THEN-ELSE branching constructs. A condition at the beginning of the first subchart will form the IF statement. If it is evaluated to be true, the rest of the first chart will be performed. The execution will jump to the exit event when the transition location is reached. If the statement is false, execution will jump to the transition event, and the second chart will

be executed. Figure 4.2 should not be viewed as a concrete example, but more as an illustration of which locations can found in an instance.

Messages are defined inside <message> tags. The sender and receiver of the message are referenced by idref attributes inside <from_instance> and <to_instance> tags. The name of the message is acquired from the body of a <name> tag. A message can have parameters which are placed between a <parameters> tag. Along with id, the <message> tag can have attributes for whether the message is synchronous or not. By default it is synchronous, but by setting 'synchronous="false"' it will be asynchronous. The temperature of the messages can be defined by a temperature attribute in the <message> tag. Valid values are "hot" and "cold".

```
<message id="" synchronous="true" temperature="hot">
    <name>____</name>
    <from_instance idref=""/>
        <to_instance idref=""/>
        <parameters>
           <parameter idref="">
        </parameters>
</message>
```

Figure 4.3: Messages

Symbolic messages use parameters. These parameters must be defined for each chart, and referenced from the messages that use them, as seen in Figure 4.3. The order of the referenced parameters reflects the order of messages parameters. A parameter definition can be seen in Figure 4.4.

```
<parameter id="">
    <name>___</name>
</parameter>
```

Figure 4.4: Parameter definition

Conditions are written inside <condition> tags, with the expression inside the <name> tag. A condition is synchronized for one or more instances, the references to the instances are obtained from the idref attribute in <instance> tags. The <instance> tags are placed in the body of a <synchronize> tag. As with messages, a condition also has a temperature, this is defined by a temperature attribute in the <condition> tag.

```
<condition id="con.1" temperature="hot">
    <name>B.active = true</name>
    <synchronize>
        <instance idref="B"/>
        <instance idref="C"/>
        <instance idref="D"/>
    </synchronize>
</condition>
```

Figure 4.5: Condition

Assignment elements are constructed in much the same way as conditions. <name>, which contains the assignment expression, and <synchronize> tags are placed in the body of the <assignment>. As assignments do not have any temperature, it is not a valid attribute.

```
<assignment id="assi.1">
    <name>X  := B.power</name>
    <synchronize>
        <instance idref="B"/>
        <instance idref="C"/>
    </synchronize>
</assignment>
```

Figure 4.6: Assignments

Subcharts can be used to create loops; this is done by setting a loop attribute in the <subchart> tag. Two different loops type are supported, limited and unlimited. For a limited loop the loop attribute is set to the numeric value that describes the number of iterations wanted. If an unlimited loop is needed, the attribute is set to an asterisk. The instances involved in the subchart must be referenced between the <instances> tags. A subchart does not necessarily have to be a loop; it might just as well be used to describe hierarchy in the chart, or to create branching constructs such as IF-THEN and IF-THEN-ELSE. In the case of an IF-THEN-ELSE construct, two different subcharts are needed, one for the IF part and one for the ELSE part. The transition between the two is described in the instance locations.

As per the LSC definition, a subchart may apply for other instances than the ones defined for the main chart. Our format does not have support for this.

```
<subchart id=”sub.1” loop=”*”>
    <instances>
        <instance idref=”A”/>
    </instances>
</subchart>
```

Figure 4.7: Subchart

Precharts are constructed in much the same way as subcharts. The instances involved in the prechart scenario are referenced inside <instances> tags, which is placed inside <prechart> tags.

```
<prechart id=”pre”>
    <instances>
        <instance idref=”A”/>
        <instance idref=”B”/>
    </instances>
</prechart>
```

Figure 4.8: Prechart

4.3 Constructing XML from a Chart

Figure 4.9 shows a LSC that we want to create an XML file for. The first thing that should be done is to locate and create the instances and place them in a <chart>. Here we find User, Volume and Power, and create three <instance> elements for them in the main chart. The name of the chart and instances should be placed in the body of a <name> tag.

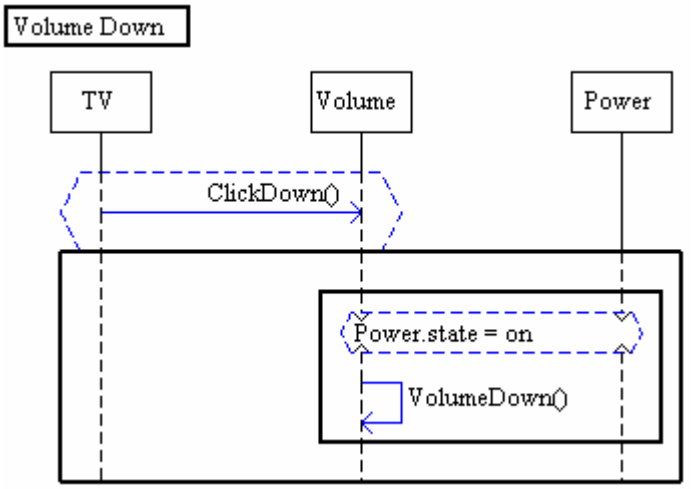

Figure 4.9: LSC for Volume Down

All elements we create must be uniquely identifiable by an id attribute. The id might be anything, but it is wise to use something that gives some clue as to which kind of element it is. For messages “msg.1”, “msg.2”, and so on can be used, for conditions “con.1”, “con.2” and so on. Instead of a number, the name of the message, instance or chart can be used. For our chart we choose the id "chart.vdown", and for the three instances “inst.User”, “inst.Volume” and “inst.Power”.

The next step will be to create the messages, precharts, subcharts and conditions. They should all be placed in the body of the <chart> tag. In Figure 4.9 we have one condition, one subchart, one prechart and two messages, ClickDown() and VolumeDown().

The instances messages are sent between should be located and their unique id should be referred to by the idref attributes in <from_instance> and <to_instance> tags in the message body. VolumeDown() is sent between User and Volume, so the idref in <from_instance> should be inst.User, and the reference in <to_instance> should be inst.Volume.

For the condition, the expression should be placed in a <name> tag, and placed in the body of the <condition> tag. Which instances the condition is synchronized for should be located and placed as references to instances in between the <synchronize> tags in the

body of the <condition> tag. The condition in Figure 4.9 is synchronized for Volume and Power, so inst.Volume and inst.Power should be referenced in the <synchronize> tag.

For the subchart and the prechart the instances involved should be identified and placed as references in between <instances> tags. In the prechart User and Volume are involved and in the subchart Volume and Power are involved.

The last step should be to determine the location of each of the instances; this is easily done by looking at the chart in Figure 4.9. Just list the events as they occur from top to bottom by creating references to the elements already created. Special emphasis should be given for references to prechart, subcharts and main charts. For precharts and subcharts, it must be specified what kind of event it is, that is, entry or exit. This is done by writing it in the event attribute in the subchart and prechart references. A main chart only has entry events, so there is no need to specify it, but the reference must be placed in the correct position. It should be placed after a prechart exit event, or as the first event for instances that are not part of the prechart. The locations of the Volume instance would for instance be: entry of the prechart, the message ClickDown(), the exit of the prechart, the entry of the main chart, the entry of the subchart, the condition, the message VolumeDown(), the message VolumeDown() again and last, the exit of the subchart. Figure 4.10 shows the XML code for the chart described in Figure 4.9.

```
<chart id="chart.vdown">
    <name>Volume Down</name>
    <instance id="inst.User">
        <name>User</name>
        <locations>
            <prechart idref="pre" event="entry"/>
            <message idref="msg.1"/>
            <prechart idref="pre" event="exit"/>
            <chart idref="chart.vdown"/>
        </locations>
    </instance>
    <instance id="inst.Volume">
        <name>Volume</name>
        <locations>
            <prechart idref="pre" event="entry"/>
            <message idref="msg.1"/>
            <prechart idref="pre" event="exit"/>
            <chart idref="chart.vdown"/>
            <subchart event="entry" idref="sub.1"/>
            <condition idref="con.1"/>
            <message idref="msg.2"/>
            <subchart event="exit" idref="sub.1"/>
        </locations>
    </instance>
    <instance id="inst.Power">
        <name>Power</name>
        <locations>
            <chart idref="chart.vdown"/>
            <subchart event="entry" idref="sub.1"/>
            <condition idref="con.1"/>
            <subchart event="exit" idref="sub.1"/>
        </locations>
    </instance>
    <prechart id="pre">
        <instances>
            <instance idref="inst.User"/>
            <instance idref="inst.Volume"/>
        </instances>
    </prechart>
    <condition id="con.1" temperature="cold">
        <name>Power.state = on</name>
        <synchronize>
            <instance idref="inst.Power"/>
            <instance idref="inst.Volume"/>
        </synchronize>
    </condition>
    <subchart id="sub.1">
        <instances>
            <instance idref="inst.Power"/>
            <instance idref="inst.Volume"/>
        </instances>
    </subchart>
    <message id="msg.1">
        <name>ClickDown()</name>
        <from_instance idref="inst.User"/>
        <to_instance idref="inst.Volume"/>
    </message>
    <message id="msg.2">
        <name>VolumeDown()</name>
        <from_instance idref="inst.Volume"/>
        <to_instance idref="inst.Volume"/>
    </message>
</chart>
```

Figure 4.10: XML code

4.4 Binding LSCs to the System Model

To allow the XML format to couple with the system model, we use a simple <bind> tag, for the message or instance that should be bound. Figure 4.11 shows an example of this. The instance is bound to the class Person, the message is bound to the method "getName" in the class it is invoked on, which is Person.

```
<instance id="id.01">
    <name>person</name>
    <bind>Person</bind>
    <locations>
        :
        :
</instance>

<message id="id.02">
    <name>getName(X)</name>
    <bind>getName</bind>
    <to_instance idref="id.01"/>
    <from_instance….>
</instance>
```

Figure 4.11: Binding example

4.5 Summary

The language described is limited in several aspects, but it should provide a simple and intuitive way of constructing basic charts. It should be relatively easy to extend it to allow for additional LSC elements. In Appendix B an XML Schema for the XML format is supplied.

In the next chapter we will describe an algorithm for building a tree structure from the events listed for instances within an XML file.

5 Algorithm for Building an Event-Tree

5.1 Introduction

Partial order controls the order of execution, and should be represented in a way that is easy for computerized tools to use. In the XML format for LSC we wanted to have a simple and intuitive way of constructing charts, with little emphasis on the partial order of events. The only description of the ordering of events is the vertical ordering of locations in each instance.

In this chapter we describe an algorithm for building an event tree based on the vertical listing of locations in an XML file. The presented algorithm is somewhat limited and does not allow for asynchronous messages. Our LSC visualization program uses this tree as information on which order the elements in a chart should be drawn.

5.2 Methodology

The first thing that has to be done is to make couples for events along an instance line. If there are three events along an instance line called A, B and C, two couples will be made, [A, B] and [B, C].

Each couple describes a minimal ordering, containing two events, a “cause” and an “effect”. The “cause” event will happen before the “effect”, but there may be several events between the two events. Couples are made for every instance, and the collection of couples is used for building the tree.

The collection of couples is searched for “minimal events”, events that have no cause. Events that are only in “cause” parts, and not in any “effect” parts of couples in the set, are “minimal events”. For each “minimal event” found, a new couple is constructed. The new couple will have “start” in the cause part of the couple, and the “minimal event” in the effect part. This is done to make a more generic tree structure, where all nodes can be traced back to one single ancestor.

Each node in the event-tree can have multiple parent nodes and multiple child nodes. A node is also associated with a level, describing its vertical position in the tree. Cyclic linking is not allowed, but the merging of branches is allowed. If a node has two children, it means that the order in which these two events occur is arbitrary. If two nodes has the same child node, it means that the two events the nodes refer to must happen before the child nodes element can be performed.

The next step is to build the actual tree with the set of element couples. First a root node for the starting element “start” is created. This is passed as the root node and the node

we want to build from in the *buildtree* method described in the pseudo-code in Figure 5.3.

After the root node has been created and passed to the algorithm, the algorithm will iterate thru the couples and look for a cause-part that is identical to the node we want to build from. A new node for the effect-part of the couple should be added to this node, but first we need to check that the tree does not already contain a node for the element we want to add. There are two ways in which the element may exist, as a direct link upwards in the tree as seen to the left in Figure 5.1 or as node on another branch of the tree as seen to the right. In Figure 5.1 the element in the dashed circle indicates the element that is to be added to the tree.

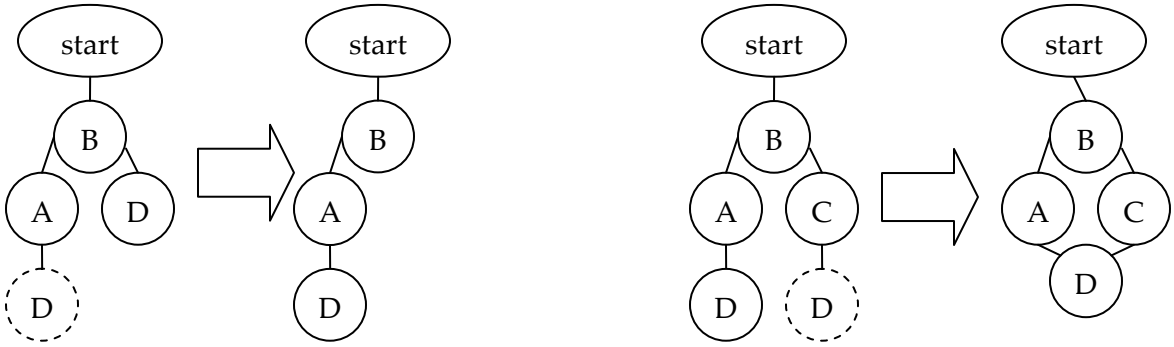

Figure 5.1: Direct Link and No Direct Link

When the element exists as a direct link upwards in the tree, the node is simply removed from the tree, along with all its children. If there is no direct link, the node can not be removed directly, but the element should be merged with the existing node we are about to add. Figure 5.1 shows these two possibilities. Once all the couples have been checked and eventual event nodes are added, the process is repeated for the events that have been added to the node we are building from. Figure 5.2 illustrates the different stages and the algorithm for building a tree with an example.

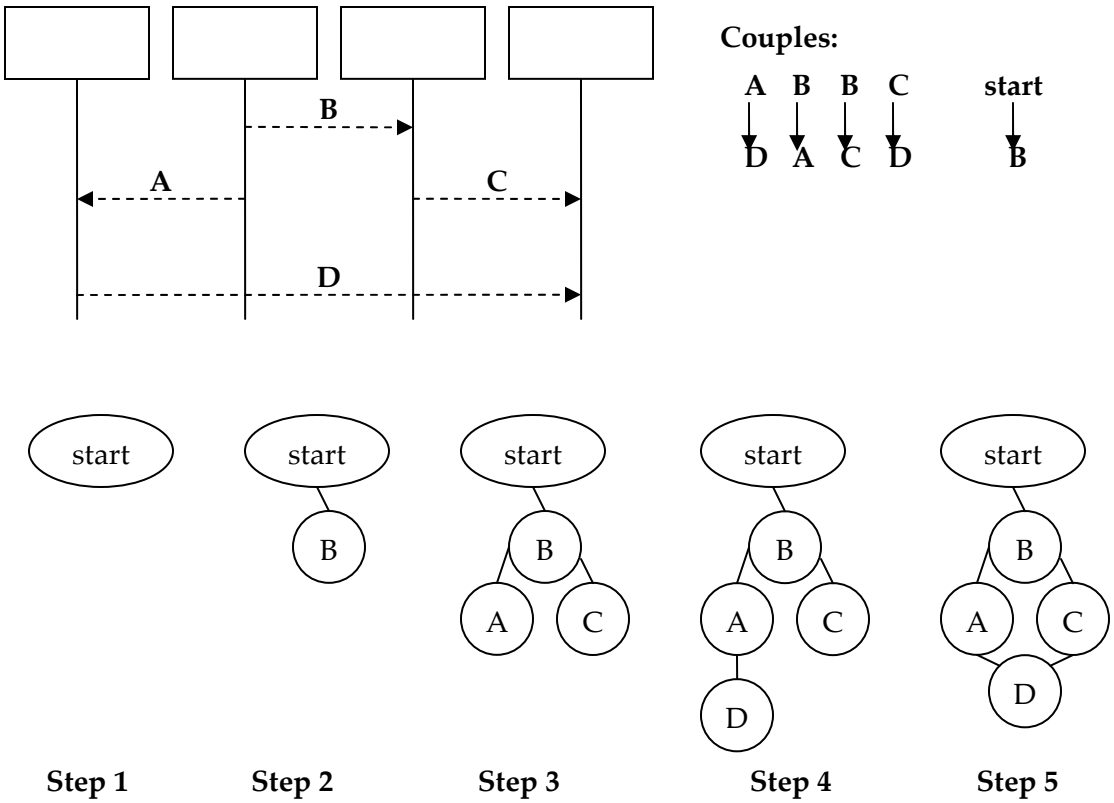

Figure 5.2: Building an event-tree

Figure 5.2 shows a simple chart, with four instances and four messages. The couples for the chart are listed with the cause-part on top and an arrow to the effect part. Event B is to found to be a "minimal event", as it is only listed in the cause row and not in the effect row. A special couple for the B is created with 'start' as the effect and B as the cause.

The building of the tree is shown in five steps. In step one the root node is created for 'start', and building is begun with this as the current node. In step two the couples are searched for couples that have the current node as the effect part. The couple [start, B] is found, and a node for B is added to 'start'. Next B is set as current element and the couples are searched for couples with B as the "cause"-part. The couples found would be the [B, A] and [B, C]. Nodes for both A and C is added to B in step three. A is the current node in step four, and D is added to A because of the couple [A, D]. In step five C is the current node, the couple [C, D] indicates that a node for D should be added to the node for C, but there is already a D in the tree. There is no direct link upward to D from C, so the D-node that already exists has to be merged with the one we want to add. This is done by setting C as parent element for D and setting D as child element for C.

5.3 Using the Event-Tree

Because a node in the tree can have multiple parent and child nodes and merging of branches is allowed, traversing the tree is not as trivial as one might expect. To build the event tree recursive methods must be used. Building sets of node elements that are on the same level of the tree can be a great alternative to recursive methods.

Our implementation for the event-tree has a *PartialOrderElement* class that contains a method *GetOnLevel(int level, ArrayList array)* that, when called on the root-element, adds all node-elements on the requested level in an *ArrayList.* The method *GetHighestLevel()* can be called on the root-element to give the number of levels to iterate thru.

```
buildTree(root, element, couples)
begin
       For each couple c in couples
       If c.cause == element then
          Remove direct links to c.effect upward in tree from element

          If c.effect still exist in tree then
             Merge c.effect-element as child to element
          Else
             Add c.effect as child to element
          End if
       End if
       Next

       For each child in element
       Buildtree(root, child, couple)
       Next
End
```

Figure 5.3: Algorithm pseudo-code for building an event-tree

5.4 LSC Visualizer

We have constructed a simple application to visualize LSCs described by our XML format called "LSC Visualizer". The application opens and parses an XML file and creates an event-tree with the algorithm described above. This tree is used to determine the order of how the events such as messages, precharts and conditions should be drawn. The application was built in C# using the Microsoft .NET Framework SDK [18].

The "LCS Visualizer" is very simplistic in its use. After a file is opened and parsed, the user can choose between two view modes, chart-view and events-view. In chart-view mode the chart is shown with the constructs and elements as described in chapter 2. The event-tree that is generated from the XML data can be seen in events-view. Every event-tree node is displayed with the id attribute from the corresponding XML element. For entry and exit of precharts and subcharts the type of event is added after the id attribute, for instance "prechart.exit". Figure 5.4 shows a screenshot of the application in events-view mode and Figure 5.5 shows a screenshot of the application in chart-view mode.

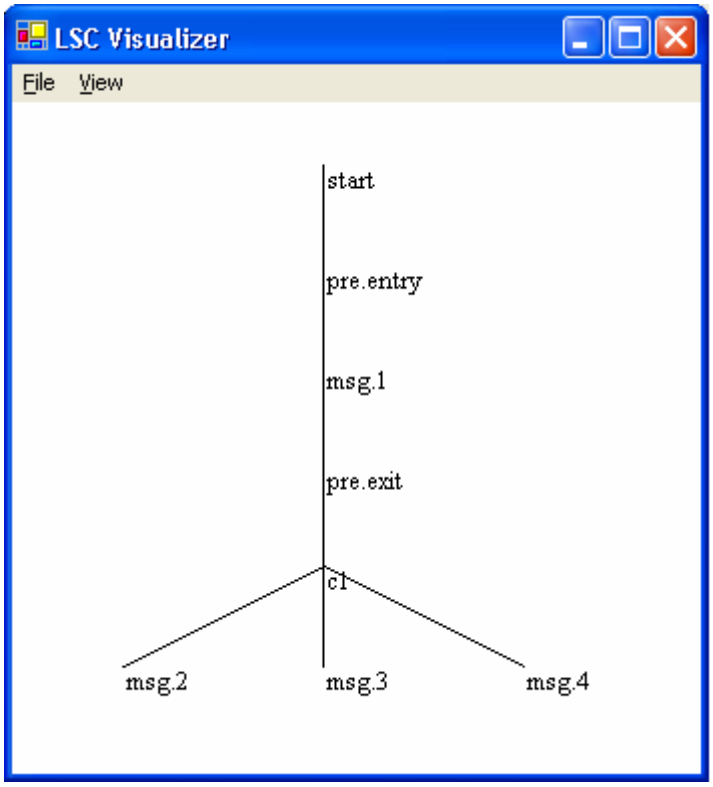

Figure 5.4: "LSC Visualizer" in Events-view mode

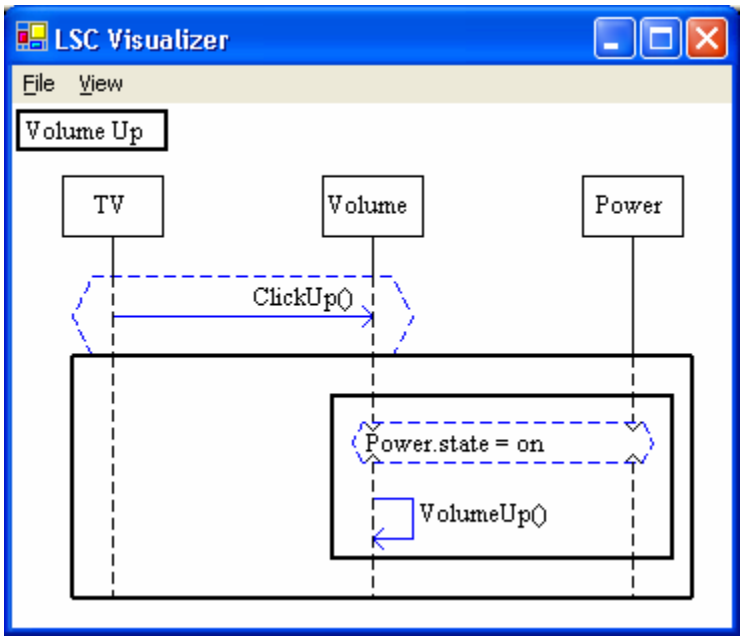

Figure 5.5: "LSC Visualizer" in Chart-view mode

It should be noted that this program was developed for use in our work with this thesis only, and is not intended to be a final version. It might not display every chart correctly,

and the application is also very sensitive to errors in the XML code. The visualizer has been used as a basis to create the LSC chart figures in this report.

5.5 Summary

We feel that the algorithm presented in this chapter is a versatile one. It has been used to determine the order to draw events in the "LSC Visualizer", and it is used in the core of the code generation solutions presented in chapters 8 and 9. Although the algorithm may look simplistic, it has proved to be sufficient for our needs throughout this thesis.

In the next chapters we will take a look at different possibilities code generation can introduce in a development process.

6 Introduction to Code Generation Possibilities

In this chapter we will present some of the basis we use for our studies in code generation from LSCs. We present a development process for the creation of complex reactive object-oriented systems which is used as a roadmap. We focus on a system model where system structure and system behavior are coupled to describe it. At the end of this chapter in Figure 6.1 we have sketched a development process outline.

In a typical software development process the client and the developer will first meet to discuss the requirements of the system. The requirements agreed upon would be the basis for use-cases describing different scenarios of the system. Behavioral requirements would be created to instantiate these use-cases. There are several approaches for modeling the behavioral requirements, like pseudo-code, the more common message sequence charts or the newly introduced live sequence charts. In this chapter we will focus on a highly LSC centric development scheme.

At first these behavioral requirement described by LSCs would typically be sample interactions that we want the system to exhibit. As the development continues, these requirements would be refined in iterations. One can look at this as going from an "existential view", where sample interactions of the system is described, to a "universal view", a more complete description of the actual behavior we want the system to exhibit.

The next step is to create the system model from the requirements. There is typically no automation in this step, but there exist step-by-step guidelines. With the expressive power of LSCs this could change, and computerized tools could be deployed to synthesize the system model or a good assessment of it.

We will focus on an object-oriented analysis and design development scheme, where the system model has two aspects, structure and behavior. The structure of the system will consist of class diagrams and statecharts are used to capture the systems behavior.

Generating code from an object-oriented system model is a well studied issue, and there exist tools that can generate code from statecharts. These tools include Rose RealTime from Rational Solutions [11], PoseidonUML from Gentleware [10] and Rhapsody from I-Logix [12].

The last step in the development scheme is the final implementation. We separate this from the code, as we are dealing with code generation; the code generated might not be sufficient and might have to be refined before it can be called a final implementation of the system.

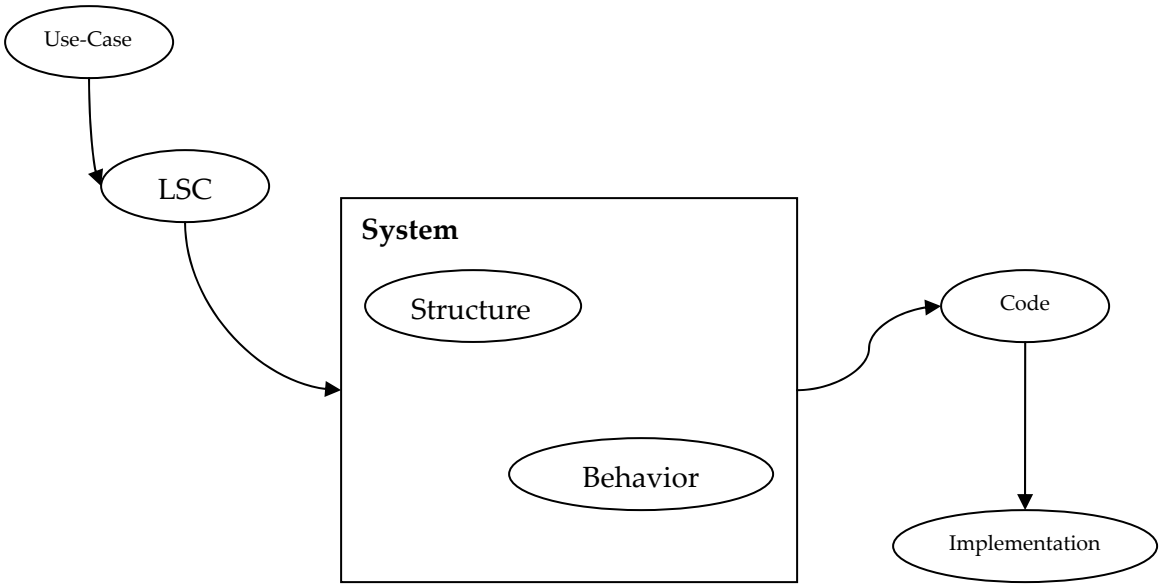

Figure 6.1: The development process

Figure 6.1 shows the development scheme we have outlined in this background. The ability to generate code can be useful at several stages in this process. In the next three chapters we will present three conceptual solution of the application of code generation.

7 Code Generation by Synthesizing Statecharts

7.1 Introduction

A part of this thesis is to study the possibilities of code generation from live sequence charts. One of these methods that will be discussed for the purpose of code generation is by going via statecharts [23] and from there existing tools could be used. The ability to do this automatically would require algorithms for synthesis. Synthesis is to determine if there exists an object system that satisfies a behavioral requirements model, and if so to generate it automatically from the model. Figure 7.1 shows the implications this approach of code generation would have on the development process described in chapter 6.

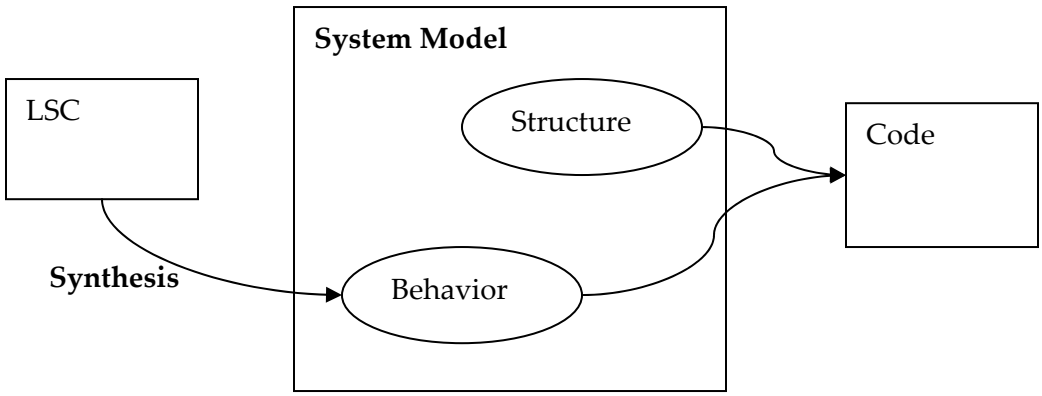

Figure 7.1: The implication of synthesis and code generation from statecharts

Statecharts are used in many UML [15] tools to model the behavior of the system. The statecharts in UML [25] are based on the statecharts introduced by D.Harel [23]. In [24] a comparison of the two is presented.

For synthesis from LSC [4] and [19] is proposed. The algorithms in [19] only deal with the generation from a single chart. Complex systems rely on the interaction of several charts to describe the behavior and [19] falls short. In this chapter we will therefore focus on outlining the ideas presented in [4].

In this chapter we will present and evaluate a methodology proposed in [4] on how to convert live sequence charts into a state based object system. First we will start by looking at the constraints needed to be in place before converting. One of the main aspects of this is making sure that the model is consistent, thus this will be a major part

of this chapter. Once the model has been proved to be consistent, one may start looking at how to convert the model into state based objects.

7.2 Method of Synthesis

One of the main reasons to convert live sequence charts into state based object systems would be to allow for code generation, by preferably using already existing tools for that task, and thus creating an implementation directly from the requirements.

Before one is able to convert live sequence charts to a state based object system however, there are certain constraints that need to be fulfilled. One must make sure that there are no self contradictions in the charts and then make sure that the LSC specification is consistent. In [4] proof is presented that if a LSC specification is consistent it implies that the specification is satisfiable. When a specification is satisfiable it is possible to construct a Global System Automate (GSA) from all the charts in the model. A GSA proves the existence of an object system, a separate automaton for each object that satisfies the system. Statecharts can be generated by distributing the GSA amongst the objects in the system.

Our TV example application will be used partially in this chapter to demonstrate some of the aspects of the converting. However, we do not expect a full conversion from live sequence charts to statecharts as there is only an outlined algorithm of the actual conversion at the time of writing of this thesis, and there are limitations imposed on the charts. The algorithms and proofs presented in [4] only apply for a subset of the LSC language. Only a single instance of each class in the system is considered. The methodology is also limited to only apply for a synchronous message model, and a model with no failures in the sending and reception of messages. Further only activation messages and no precharts are considered as scenarios for when universal charts should apply. The methodology does not mention conditions or subcharts, which are elements our Television example depend upon.

7.2.1 Satisfying the LSC specification

A specification is satisfiable if and only if the system is consistent. There has been developed an algorithm in [4] for the purpose of determining if the system is consistent. If the output of that given algorithm is YES, the system has been determined to be consistent.

In short the algorithm works like this. Build an automaton for each universal chart in the specification and intersect these to create the “largest” regular language satisfying all the universal charts. Then narrow the intersection down to avoid states from which the environment forces a system violation. When this is done the automaton needs to be

checked against the existential charts to see if there are any representative runs satisfying each of them.

There are two reasons why this algorithm may give NO as an output. If there exist an existential chart such that there is an inconsistency between that chart and either of the universal charts, the result will be NO. The other possibility is if there is a sequence of messages sent from the environment to the system that causes a situation where one of the universal charts can not be satisfied.

There is also the possibility of having contradicting charts, meaning there are two or more charts that operates on the same two or more messages, but have a different partial order. Let us consider message M2 and M3 and chart C1 on the left side and C2 on the right in Figure 7.2. If the message M0 occurs in the system, C1 will be activated. This will force the message M1 to be sent between instances in C1 and also activate C2 and force the execution of that chart as well. If C1 expects M2 and then M3 as being sent while C2 expects M3 and then M2. These charts will contradict each other and thus break the consistency.

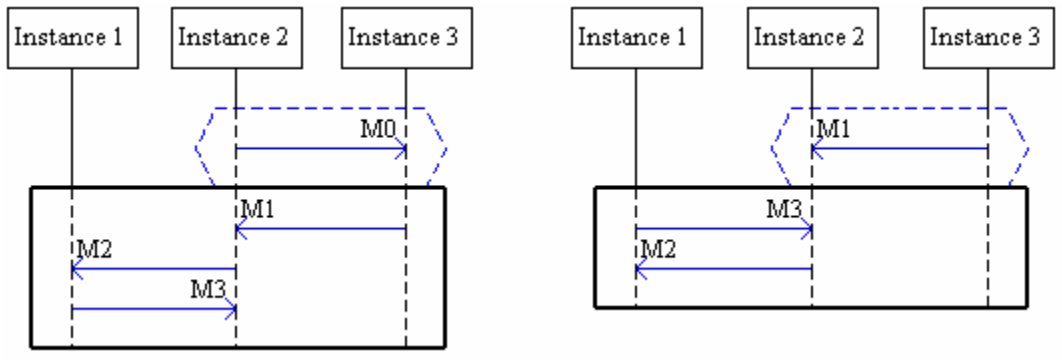

Figure 7.2: Contradicting charts

Another source of self contradiction in a system might arise from specifications where the restrictions of universal charts exclude the possibility of any successful trace of one or more existential chart.

When the system has been determined to be consistent, one may start looking at how to convert the live sequence charts into a statecharts. First we start be building a global system automaton (GSA).

7.3 Constructing the Global System Automaton

Each and every universal chart can be converted to an automaton. The collection of these automatons will make up what is referred to as the intersection automaton. The global system automaton is the automaton where both the universal charts and runs satisfying

the existential charts are considered. An automaton describes the behavior of a single chart, thus the global system automaton describes the behavior of the entire system.

As an example of creating a global system automaton we start by looking at the "TV on/off" chart shown with marked locations in Figure 7.3. We also define a set of messages that are restricted for the chart, this is the set of messages not appearing in the chart: { VolumeUp(), VolumeDown(), ChannelUp(), ChannelDown(), ClickUp(), ClickDown() }.

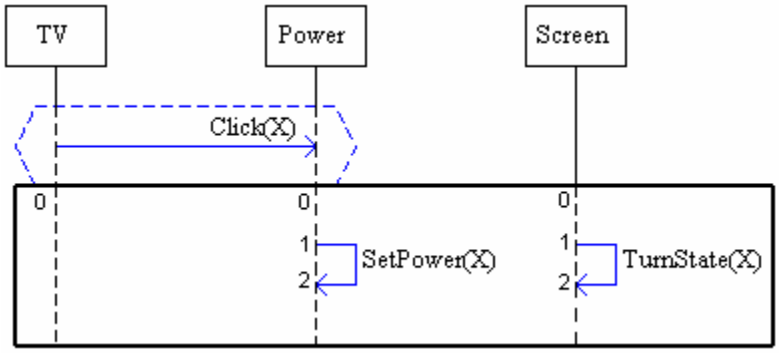

Figure 7.3: "TV on/off" chart with marked locations

The different cuts and runs of the charts must be identified before an automaton can be constructed. More information about cuts and runs can be found in section 2.5 and 2.6. The cuts for the main chart for "TV on/off" are:

```
{ (<TV, 0>, <Power, 0>, <Screen, 0>), (<TV, 0>, <Power, 2>, <Screen, 0>),
  (<TV, 0>, <Power, 0>, <Screen, 2>), (<TV, 0>, <Power, 2>, <Screen, 2>) }
```

The chart "TV on/off" has two possible runs, one when SetPower(X) occurs first, and one when TurnState(X) occurs first. Figure 7.4 shows an illustration of the runs of the chart. This illustration is used to construct the automata in Figure 7.5. The self references in the states allows for messages that are not restricted by the chart.

SetPower(X) first:

```
{ (<TV, 0>, <Power, 0>, <Screen, 0>),
  (<TV, 0>, <Power, 2>, <Screen, 0>),
  (<TV, 0>, <Power, 2>, <Screen, 2>) }
```

TurnState(X) first:

```
{ (<TV, 0>, <Power, 0>, <Screen, 0>),
  (<TV, 0>, <Power, 0>, <Screen, 2>),
  (<TV, 0>, <Power, 2>, <Screen, 2>)}
```

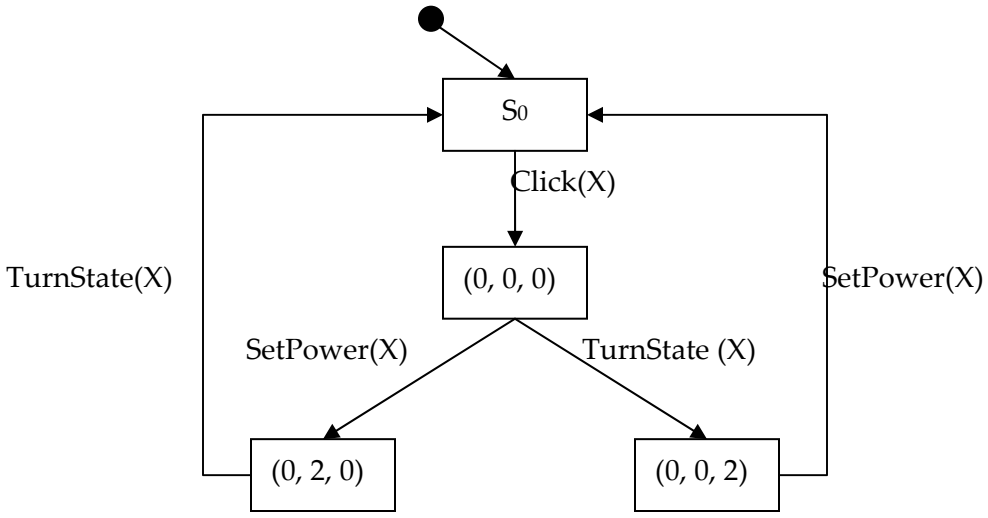

Figure 7.4: Automaton representation for all possible runs

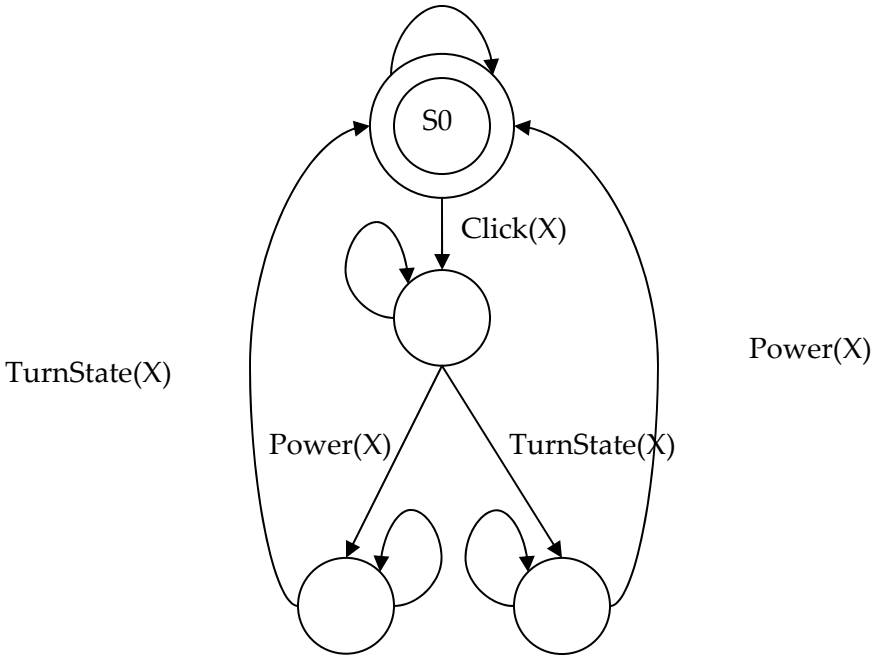

Figure 7.5: Automaton of TV on/off

The automaton for "Volume Up" is shown in Figure 7.6. Because of similarities between the "Volume Up", "Volume Down", "Channel Up" and "Channel Down", the rest of the

charts have been omitted. It should be noted that the algorithms and proofs in [4] does not mention subchart and conditions, which are used in "Volume Up".

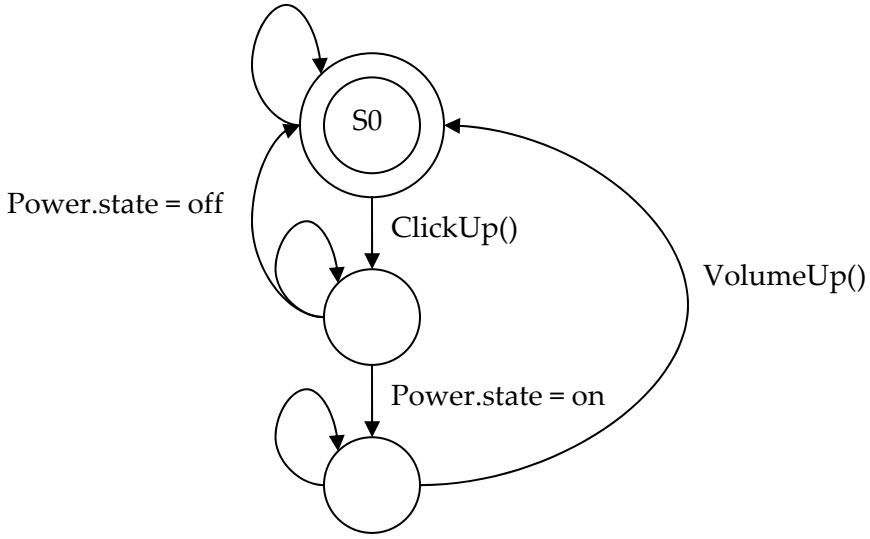

Figure 7.6: Automaton of Volume Up

If we merge the automata the result will yield the global system automaton. The GSA is shown in Figure 7.7, again "Volume Down", "Channel Up" and "Channel Down" have been omitted because of similarities.

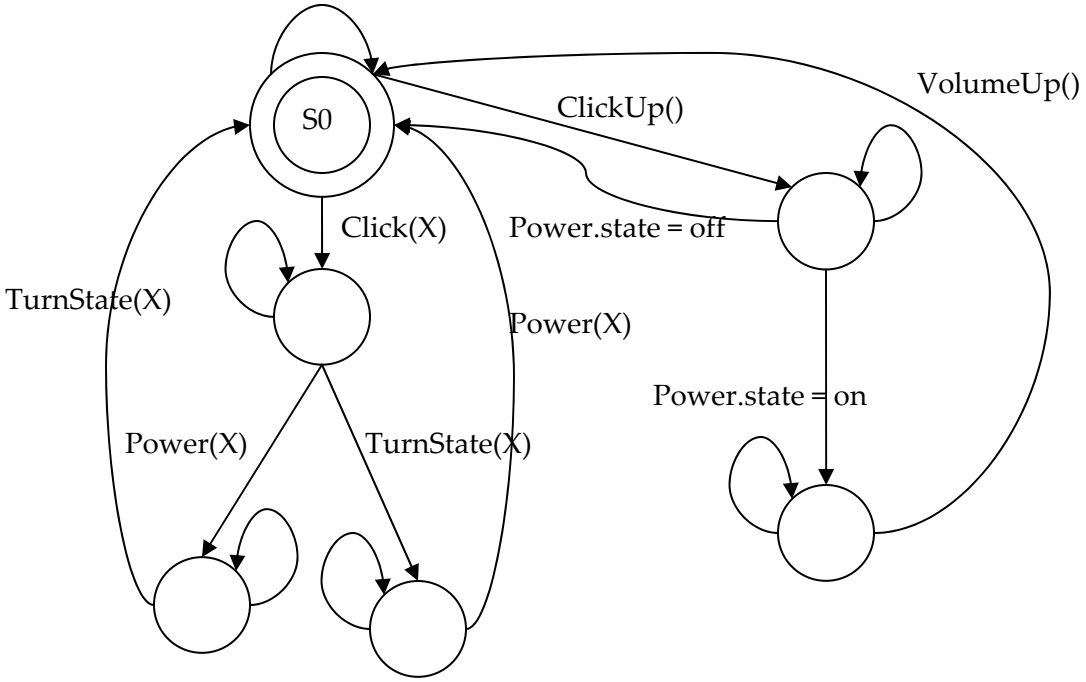

Figure 7.7: Global system automaton

7.3.1 Distributing the Global System Automaton

The next step is to create a state based object system. This is done by distributing the GSA over the objects. We will extract a subautomaton from the GSA to illustrate the points of this distribution. A subautomaton can be seen as a section of the GSA. Figure 7.8 shows a subautomaton from the GSA, the state transitions is written with sender object and receiver object and message.

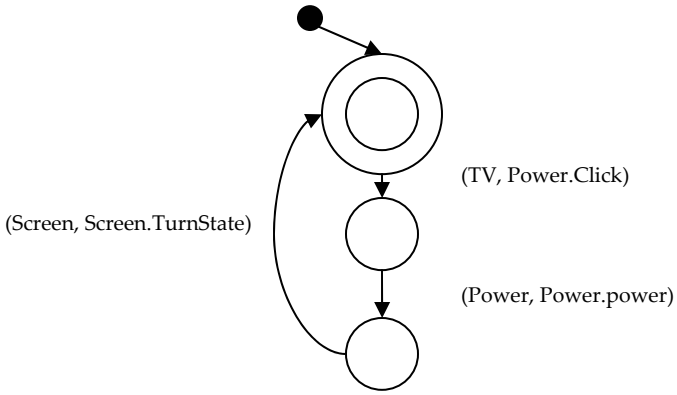

Figure 7.8: Subautomaton from the Television GSA

The three main approaches for distributing the global system automaton between the objects of the system presented in [4] are by using a controller object, by using full duplication or by using partial duplication. We will start by looking at the most trivial one, controller object.

7.3.2 Controller Object

In this approach, a controller object is added to the set of objects in the system. This object will then act as controller, which sends messages to the other objects. These objects will then have a simple automaton that enables them to carry out the commands. The size of the controller automaton will be of the same size as the global system automaton.

Figure 7.9 shows an example of the distributed subautomaton with a controller object. The simplistic automaton for the Power object can be seen at the top right side. The box on the lower right side indicates how the other simplistic automata will look.

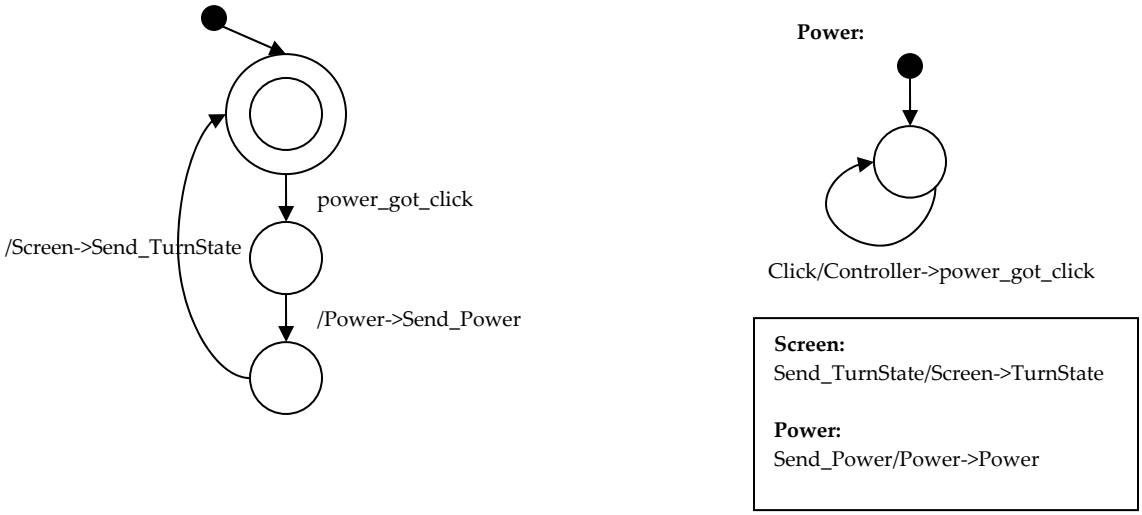

Figure 7.9: Subautomaton with controller object

7.3.3 Full Duplication

This construction contains no controller object, thus each object will have the state structure of the global system automaton, meaning the construction would know what state the GSA is in. Full duplication is a step towards partial duplication which is the most realistic approach.

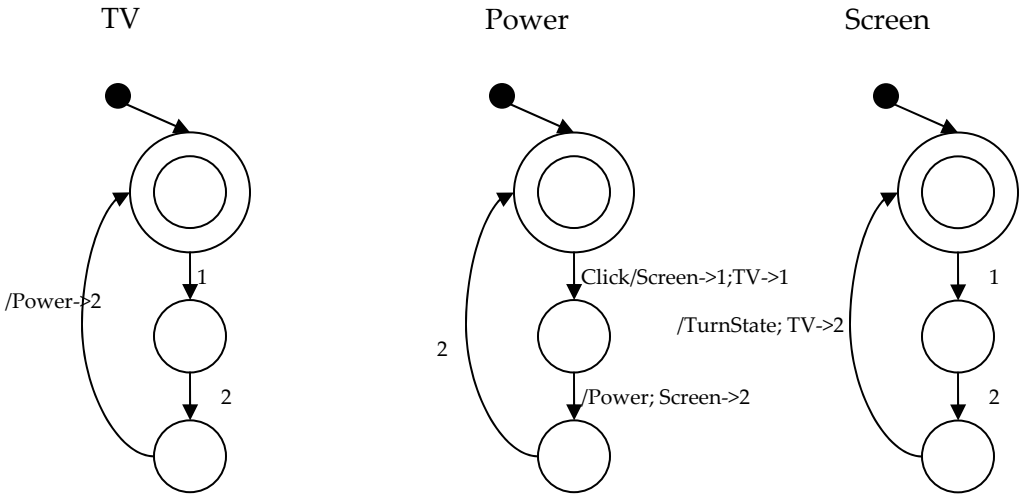

Figure 7.10: Fully distributed subautomaton

7.3.4 Partial Duplication

This approach aims to distribute the global system automaton as the full duplication method do, but it will merge states that carries information that is not relevant to the object in question. This allows for a reduction in size in most cases although the complexity can remain the same as with full duplication.

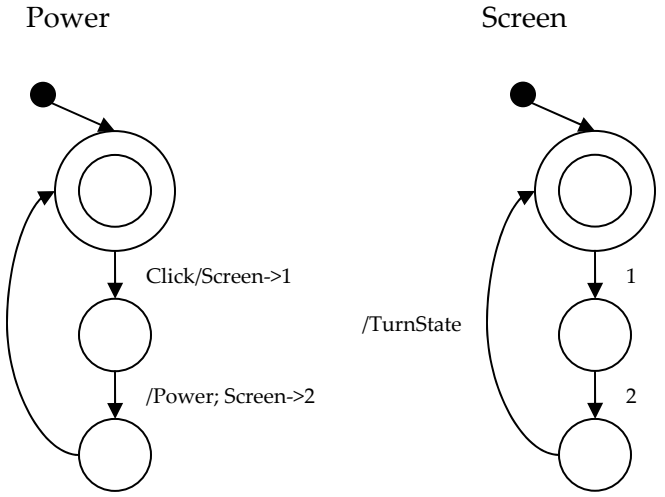

Figure 7.11: Partial distributed subautomaton

7.4 Synthesizing Statecharts

Statecharts was introduced by D. Harel in [23] as an effort of rectifying certain problems with the current state diagrams. State diagrams had a tendency of getting more chaotic as the systems got more complex. The constraints set for the statecharts was that they needed to be modular, hierarchical and well-structured.

Some of the key elements that statecharts was set to cater for were that they would allow clustering states into a superstate. They introduced orthogonality which allows for independency between states and opened up for the possibility of having general transitions and refinement of states.

The concept of generation of statecharts from live sequence charts is based on the idea of a state based object system. The initial step when going from live sequence charts to statestarts is checking consistency, then building a global system automaton which captures the behaviour of the system, then distributing it over the objects. This has been shown for a subautomata in section 7.3.1, but the methods presented does not take

account for the features and restrictions of actual statecharts. For synthesising the actual statecharts only an outlined algorithm can be found in [4]. This algorithm is based on the main succinctness feature of statecharts, presented in [23].

An object that does not actively participate in a universal chart does not need a component in the statechart. Some state configurations need to be avoided as they can cause contradictions in the statecharts or they can lead the system into a state where it will not be able to complete one of the universal charts. For the purpose of avoiding these, supercuts have been introduced. A supercut is a cut across all the universal charts in the system. A set of bad supercuts which the system should avoid to enter is calculated, and before taking a transition the system should check whether it would lead to one of the bad supercuts before taking it.

Because of the complexity of live sequence charts, they will often depend on the statecharts ability to sense other objects states and this would be a needed step before able to take a transition. If an object is in a state corresponding to a location before sending a message it should check if the receiving object is ready for reception before taking the transition and sending the message.

One of the main reasons for converting live sequence charts to statecharts would be to allow for the possibility of automatic code generation; the next section will take a closer look at code generation from statecharts.

7.5 Generating Code from Statecharts

While going from a system to a full implementation can be a formidable task, automatic code generation could benefit the developers in several aspects. It is less time consuming and the code is less error prone as it alleviates the need for manual intermediate steps where errors can occur. Automatic code generation from statecharts is possible and would greatly reduce the time needed for writing the actual code and thus the cost of developing a system could be lower. Statecharts are coupled with system structure in the form of class diagrams, and by interpreting both of these aspects it is able to generate state based code. This work is typically done by different sort of tools.

Today there exist a few tools to generate code from statecharts; among these is Rhapsody from I-Logicx [11], Rational RealTime from Rational Software [12] and PoseidonUML SE/PE from Gentleware [10] with a state to Java plug-in. The only tool available to us has been an evaluation version of PoseidonUML with a beta version of the statechart code generation plug-in. We tried to generate code from an example that came with the plug-in, although PoseidonUML was able to generate code from it, the code would not compile. We also tried to generate code from the Television example with the distributed subautomaton, but the result was the same.

7.6 Summary

While having the possibility to be able to convert live sequence charts into statecharts would be a very desirable thought, it can prove to become very difficult to do so. The consistency checking is comprehensive but useful as it allows one to determine if there exists a system that satisfies the specification. Implementing a tool that is able to take live sequence charts and convert them to statecharts, which take into consideration the constraints might prove to be very difficult to make. The methodology presented in [4] focus mainly on high level algorithms and proofs. No simplistic step by step procedure that is easy to adapt by computerized tools is presented.

As the papers available to us only outline a draft of an approach for generating statecharts from live sequence charts, it has proved to be a very difficult task to convert either of the charts the TV example has. The outlined approach only deals with a system with synchronous messages between instances with no other elements of constructs whatsoever. The charts used in the TV example require messages to be able to cover self-transitions and use synchronized conditions and subcharts.

This chapter has covered and discussed one of the suggested ways to generate code from live sequence charts. The next chapter will outline a solution for integrating a "play-out" engine into an existing partial implemented system model. The use of "play-out" will allow for the execution of charts directly and thus render the partial implementation operational.

8 Code Generation for Execution of LSCs

8.1 Introduction

It can be argued that since the "play-out" method aims to be equal to running a full implementation of the system, for some kinds of reactive systems it can in fact be used as a satisfactory implementation. This approach requires that parts of the system model are already implemented, like database communication methods or a rudimentary GUI that generates events when pressing buttons. In this chapter we will explore these possibilities. The ability to run an application before all behavior is implemented can also be a great tool for testing both application and requirements. It allows us to see if the partially implementation is behaving like it should and if the specified requirements are what we wanted.

Figure 8.1 shows the implication of a "play-out" engine in the development process described in chapter 6. The behavior is written in quotation marks to indicate that this is only a partial implementation.

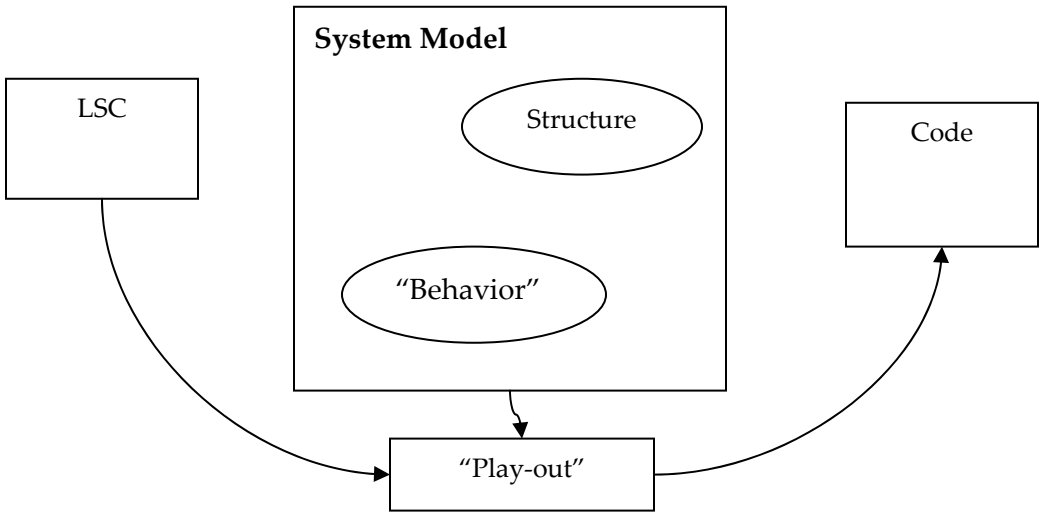

Figure 8.1: The implication of "play-out" on the development process

We have focused on generating code into a Java system model. Java virtual machines are being implemented in modern cellular phones and other mobile units and electronically equipment; these are exactly the kinds of reactive systems where code generation from reactive requirements might have an impact.

A sketch for the solution approach can be seen in Figure 8.2. Two elements of the application are needed, an "empty" implementation of the systems GUI or similar, and

the LSC model describing the behavioral requirements and inter-object communication. There has to be some connection between these two elements, our XML format for LSC described in chapter 4 supports binding to a system model.

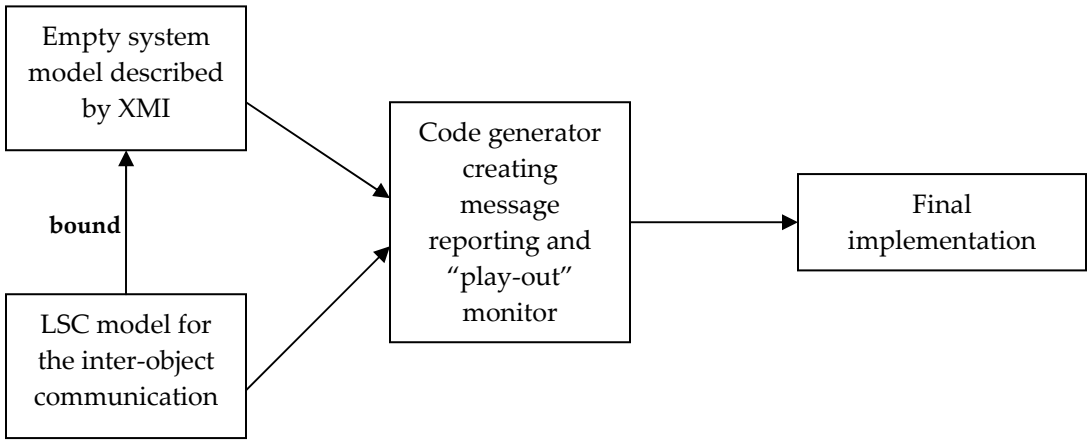

Figure 8.2: Solution sketch

These two parts of the model are merged in the next step, where message reporting is integrated in the system model. Messages are reported to a monitor that is connected to a "play-out" engine. With the use of the reported messages the "play-out" engine will look for the occurrence of minimal events in the LSC models precharts, and trace them until the precharts is successfully performed and the main chart is activated, or the prechart is violated and gracefully exited. If the prechart is successfully traced the "play-out" engine will drive the execution of the universal main chart by calling methods and evaluating conditions in the system model. If the "empty" system model and the charts are satisfactory described, and thus yielding the intended behavior of the application, it can conceivably be used as an adequate final implementation.

In this chapter we will describe the approach of generating message reporting in the system model and the implementation of a "play-out" engine more in depth. We will use the Television example throughout the chapter and show how a final implementation is generated.

8.2 Television Example

We create Java classes for the TV example introduced in chapter 2. These classes will constitute an "empty" implementation of the TV GUI, which is the reactive system we intend to model. By an "empty" implementation we mean that the core functions are present, buttons and window are created, but that the behavior is not described. One can click the buttons, but nothing happens, events are sent, but no one is listening. Figure 8.3

show the UML class diagram for the TV and Figure 8.4 shows a screenshot of the TVs graphical user interface.

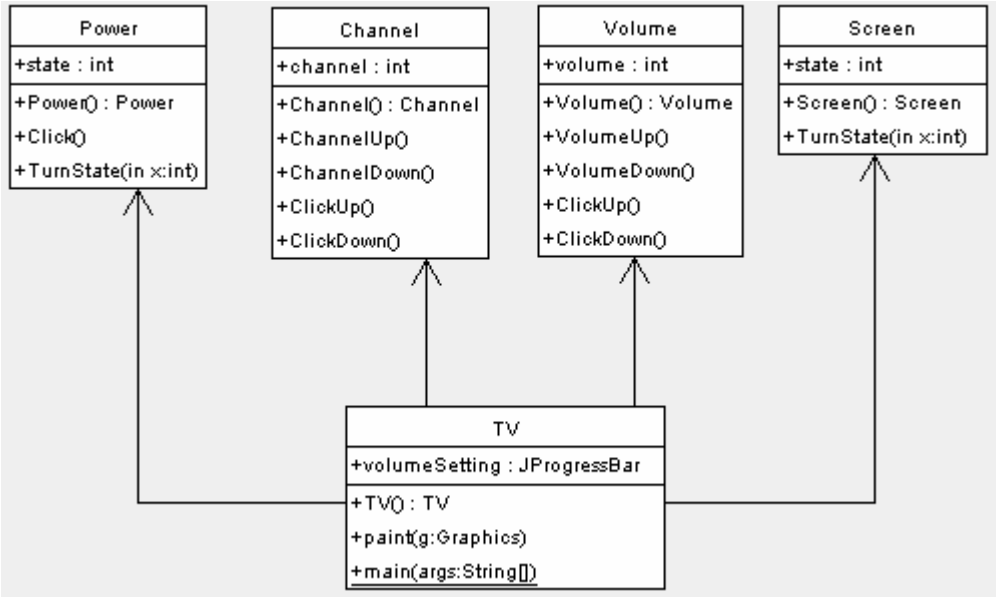

Figure 8.3: UML diagram for TV-example

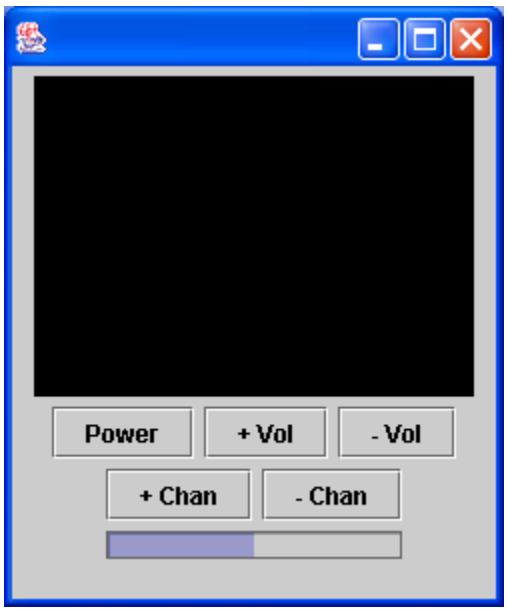

Figure 8.4: The Television example GUI

8.3 Message Reporting

The system model must be enriched with message reporting, so that a monitor can detect the occurrence of message events in the system model. A message has one sender and one receiver. This along with the message name and the parameters in the case of symbolic messages is what is interesting for the monitor. Method calls in Java will be the equivalent of messages, and the sending and receiving instances, along with the actual method name and the parameters are what should be reported to the monitor. To do this the system model would have to be changed somewhat. There are several approaches to achieve message reporting.

One possibility is to change the method itself to include a reference to the sending instance. The method would then report itself to the monitor. This alteration can be seen in Figure 8.5, the original code is to the left, and the generated code can be seen to the right with the modifications in bold characters. The monitor itself and the reason for why it is created for each class will be discussed in section 8.5.

```
class A
{

    B b = new B();

    void opA()
    {
        :
        b.opB();
    }
}

class B
{


    void opB()
    {
    // implementation
    }

}
```

```
class A
{
    Monitor monitor = new Monitor(this);
    B b = new B();

    void opA()
    {
        :
        b.opB(this);
    }
}

class B
{
    Monitor monitor = new Monitor(this);

    void opB(Object sender)
    {
        monitor.report(sender, this, "B.op", params);
        // implementation
    }
}
```

Figure 8.5: Message reporting with self-reference

Another approach is to have reporting both when the method is called and in the method itself. This would require the monitor to be able to couple the sending and receiving of messages, or calling of methods, into one event. This approach would look like Figure 8.6.

```
class A
{
    B b = new B();

    void opA()
    {
        :
        :
        b.opB();
    }
}

class B
{

    void opB()
    {
        // implementation
    }

}
```

```
class A
{
    B b = new B();
    Monitor monitor = new Monitor(this);

    void opA()
    {
        :
        monitor.report_calling(this, "B.opB");
        b.opB();
    }
}

class B
{
    Monitor monitor = new Monitor(this);

    void opB()
    {
        monitor.report_called(this, "B.opB", null);
        // implementation
    }
}
```

Figure 8.6: Message reporting from both sender and recipient

If one could find the caller of a method from within the method itself, message reporting would be simplified. This could be achieved by looking at the stack trace from the called function. In Java this is a possibility, but only the class that called the method can be found, not the instance itself. The Java documentation also states that the method that would have to be used, *getStackTrace* from the class *java.lang.Throwable*, might be incompatible with some implementations of the Java Virtual Machine. For Java one of the first two approaches must therefore be used, but for other languages, looking at the stack might be possible and preferable.

In our work we chose to use the second method of message reporting, where reporting is done before the method is called and from the method itself. Using this approach the methods themselves does not have to be changed, and it allows for future support for asynchronous messages which can occur in applications with multiple threads.

8.3.1 Parameter Passing

Eventual parameters are passed to the monitor when reporting the calling of a method. All parameters that are to be passed to the monitor must inherit *Object* class. This must be done in order to be able to add the parameters to a *Vector*. From this *Vector* an array containing all the parameters will be obtained and passed to the monitor. This method of parameter passing will exclude parameters of the primitive types, like *int* and *boolean*. Figure 8.7 shows sample code with parameter passing generated.

```
public void Click(Integer x)
{
	java.util.Vector report_vector = new java.util.Vector();
	report_vector.add(x);
	monitor.report_called(this, "Power.Click", report_vector.toArray());

}
```

Figure 8.7: Parameter passing

To generate code for message reporting, the system model can be described in XMI. A modified XMI code generator can then be used to produce code with the message reporting integrated. XMI is [17] "XML Metadata Interchange" and is used to describe an UML model in an XML format.

8.4 XMI Code Generator

We use an XMI Code generator we have created in a previous project [20] as a basis for the generation of message reporting in the system model. This XMI Code generator creates a tree structure for all the packages, classes, methods and parameters in the model. This tree structure is traversed when generating the code, and that way it is only the part that generates code for the methods that has to be changed. Figure 8.8 shows a sample of the XMI element structure. XMI refers to a method as an operation, and the body of the method as a method.

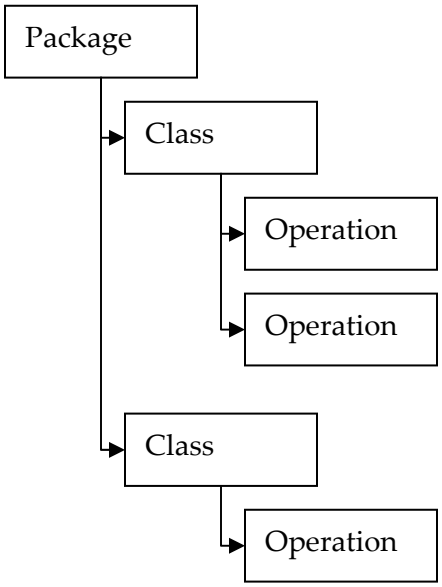

Figure 8.8: XMI Element structure

The tree is traversed and for each operation element found, code will be added for reporting that the method has been invoked. The body of the method will also have to be searched for the invocation of methods that must be reported before they are called. Only methods that are bound in the LSC specification corresponding to the system model should be called. In our approach this is done by building an index of all messages bound in the XML file and checking if a message exists in this index before reporting it.

8.5 Monitor

The monitor receives reports of methods being called and identifies message events based on this information. Since report of method calling is done before the method is called and from the method itself, the monitor must be able to combine the two into a single event. Our approach is to assume a synchronous message model, meaning that no other event can occur between the sending and reception of a message.

The monitor will receive a report of the invocation of a method and it assumes that the next report it receives is that the correct method has been called. This is not thread-safe, because a context switch can occur between the two reports.

```
class Monitor
{
    static Message current = null;
    static PlayOut playout = null;
    static Vector instances = null;

    void report_calling(Object caller, String method)
    {
        current = new Message();
        current.from = caller;
        current.name = method;
    }

    void report_called(Object called, String method, Object [] parameters)
    {
        if(current.name==method)
        {
            current.to = called;
            current.parameterlist = parameters;

            // check the current message against precharts
            playout.check(current);
         } else { /* error */ }
     }

    Monitor(Object o)
    {
        if(o!=null)
        {
            if(instances==null) instances = new Vector();
            instances.add(o);
            if(playout==null) playout = new PlayOut("….");
        }
    }
}
```

Figure 8.9: Monitor code extract

The monitor must be globally accessible from all classes in the system model. We chose to do this by having all the classes in the system model instantiate different monitors, but have all the different monitors work on the same static message, while keeping the methods non-static. The monitors will also have to keep track of the different instances in the system model; this is done so that the "play-out" engine can access them. When each of the instances in the system instantiates the monitor they pass a self reference, as shown in Figure 8.6. The reference is added to the register of the monitor in the constructor. This register along with the "play-out" engine itself will also be static so that there will only exist one globally unique instance of them. The monitor has to be generated specifically for each system, as it initializes the "play-out" engine with the actual XML file describing the LSC model.

Once a message event has been identified it should be checked against precharts that should be activated or against already activated precharts that might progress because of the message event. Information of precharts should be gathered from an LSC specification described in XML that is specifically coupled with the system model as described in section 4.4.

8.6 "Play-Out" Engine

We have developed a simple "play-out" engine for use by the monitor. This is not a full implementation of the methodology described in [2], but it should suffice for small applications or systems and be robust enough for a "proof-of-concept" model for this approach to code generation.

The "play-out" engine contains the LSC model, which is loaded from file when the "play-engine" is initialized from the monitor. The given file is passed to the engines constructor. The "play-out" engine is implemented as a thread that should run alongside the main program.

The LSC model will consist of the collection of live sequence charts that make up the behavioral requirements. For each of these charts the algorithm described in chapter 5 is used to build a tree describing the partial order of events within the chart. This tree is used to track progress in the chart. A *boolean* value for each node in the tree indicates if the LSC element the node references has been traced. An event is traced if it has been located in the prechart or if it has been executed or evaluated in the main chart. Figure 8.10 shows a UML diagram with extracts of the "play-out" engine. The extracts has been made for easy reading, and to emphasis details of the "play-out" engine described in this chapter.

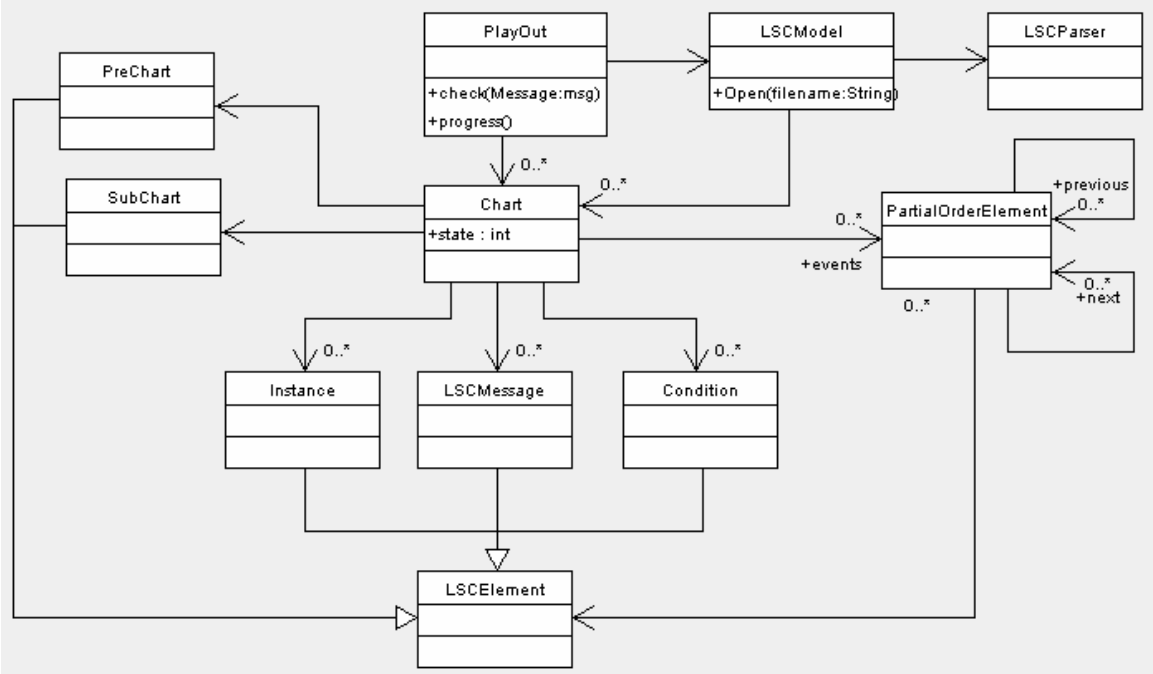

Figure 8.10: The "play-out" engine

When the "play-out" engine receives a message from the monitor it will be checked against the LSC model as described in the pseudo-code in Figure 8.11. Preactivated charts will be copied into an array of charts that is maintained by the "play-out" engine. The event-checker will also check the events against charts that are pre-activated. If the event violates the prechart the chart will simply be deleted from the copy-array, but if it forces the prechart to progress and be traced successfully, the chart will be marked as Activated, and the universal part of the chart will be executed. Parameters in symbolic messages will be set from the parameters of the checked messages if they match a LSC message that uses symbolic parameters.

```
checkEvent(Message m)
begin
    for each chart in the lsc model
        check m against the charts minimal events
        if m is a minimal event then
            copy chart into copy-of-charts-array
            set copy of chart to PreActiveted
        end if
    next

    for each chart in copy-of-charts-array
        if chart is preactivated then
            check m against untraced elements
            if prechart traced successfully then set copy of chart to Activated
            if m violates prechart then delete chart
        end if
    next
end
```

Figure 8.11: Pseudo code for message checking

Charts in the copy-array that are activated will be executed. The event-tree keeps track of the progress of this execution. The "play-out" engine is run as a thread and the method *progress* described in pseudo code in Figure 8.12 will be called periodically. In our implementation we do not actually enable events and pick one of the enabled events to execute. The first event that is not traced in the event-tree is chosen for evaluation and executed if necessary.

```
progress()
begin
    for each chart in copy-of-chart-array
        if chart is in activated state then
            execute and evaluate enabled events

            if chart executed then
                delete chart from copy-of-chart-array
            end if
        end if
    next
end
```

Figure 8.12: Pseudo code for universal chart progress

8.6.1 Executing Message Events

When a message event is to be executed, that message can be an event in a prechart of a different chart and it must therefore be reported to the monitor when it is to be called. A message has references to the sending and receiving instances of the message, and when calling the method on the receiving instance, it must first "emulate" that the method is called by the sending instance. As with the classes in the system model, the LSC message class must also instantiate a monitor, but it should not send an instance reference in the same way that system model classes do. The instance reporting is circumvented by passing a *null* reference.

```
public void Execute()
{
    Class to_class = toinst.getClass();
    Method[] invokeMethods = to_class.getMethods();

    for(int i=0;i<invokeMethods.length;i++)
    {
        Method invokeMethod = invokeMethods[i];

        String methodName = invokeMethod.getName();

        if(methodName.equals(bind))
        {
            monitor.report_calling(frominst, toinst.getClass().getName() + "." + bind);

            try
            {
                invokeMethod.invoke(toinst, null);
            } catch(Exception e) { Output.println("Error calling"); }
        }
    }
}
```

Figure 8.13: Executing an LSC message

Figure 8.13 shows the code for how a message is executed. All the methods of the receiving instance are searched for the specific method that the LSC message is bound to. If and when it is found, report calling is done before the actual method is called.

8.6.2 Evaluating conditions

A "hot" condition that is violated will force the execution to halt, while a "cold" condition however can be used to form branching constructs, and must therefore be handled differently. During execution, when subcharts and main charts are entered, they are placed on a stack local to the chart and removed when they are exited.

When a "cold" condition is encountered and evaluated the path of execution will depend on the element on top of the local stack. If a condition is true, the element node corresponding to the condition will simply be marked as traced and the execution will continue with the child element nodes. If however the condition is evaluated to be false, the condition and all element nodes that spawn from the condition until the exit of the subchart that is on the stack will be marked as traced. This will force execution to circumvent these nodes. If the top of the stack is a main chart, all nodes in the chart should be marked as traced, and the chart should be exited. An illustration of branching in a chart is shown in Figure 8.14.

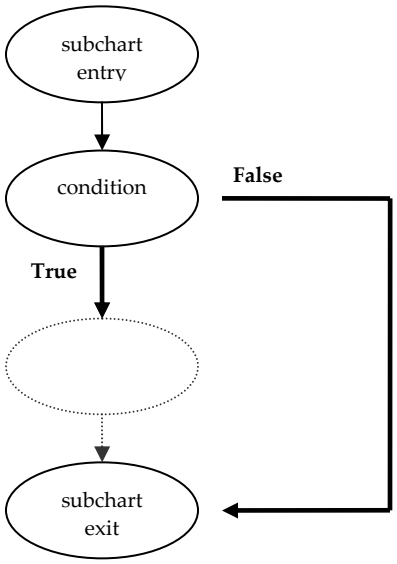

Figure 8.14: Branching during execution

8.6.3 Limitations

Our "play-out" engine implementation is limited in several ways. First of all we assume only one instance of each class in the system model. The instances in the XML file are bound to the class name. Further only conditions, synchronous messages and universal charts are supported. There is no form of event unification in the "play-out" engine; this is no problem as no similar events exist in the different universal charts of the Television example.

8.7 Result

After compiling the altered Java files with message reporting together with the monitor, the system can be run together with the classes for the "play-out" engine and the XML file for the LSC specification. The result of this is a working Television application that can be turned on and off, and with the ability to change volume and channel. Instructions on how to use the "play-out" prototype can be found in Appendix A.

In Figure 8.15 an output of the "play-out" engine running alongside with the Television application is shown. A user first tries to press the button for increasing the volume. The monitor intercepts the message and the "play-out" engine checks it against the prechart in the loaded model. It matches one such prechart, the corresponding chart, "Volume Up", is copied and the prechart is successfully traced. The main chart will then be processed. As the figure shows, a subchart is entered and then the condition "Power.state = on" is evaluated. The television is off so this will be evaluated to be false, the subchart is therefore exited, and the chart will reach maximal locations, exit and be deleted. Next the user clicks the power button. The "TV on" prechart is traced successfully and in the main chart methods for setting the power on and turning the screen on is called. The user then clicks the volume up button again. This time the "Power.state = on" will be true and the message for raising the volume will be sent.

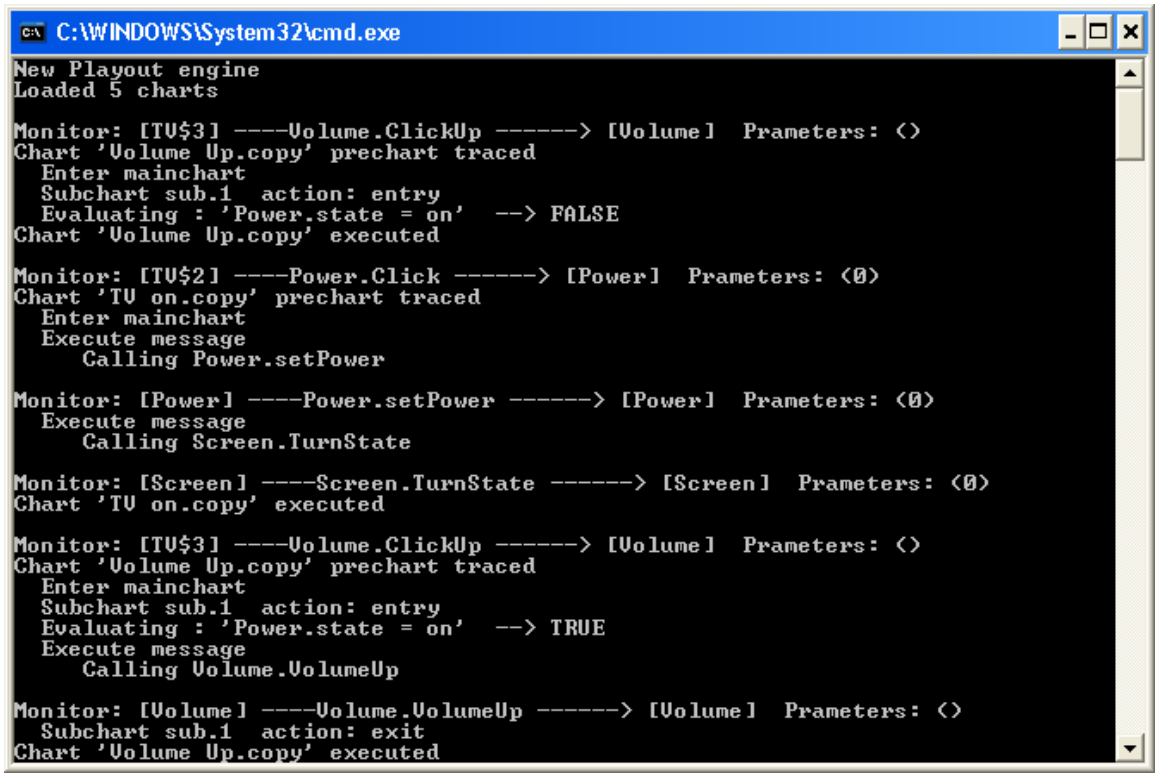

Figure 8.15: "Play-out" engine output

8.8 Summary

The main advantage of using a "play-out" engine is that it can accommodate for the full expressive power of the LSC language. We have developed a "play-out" engine prototype to illustrate the chief concepts from the "play-out" methodology described in chapter 3, and how these could be integrated with an existing system model.

The amount of code generated into the system model is minimal. There is also no directly usable code generated. This indicates that the resulting product could only be used as the final implementation of the system, or as a means of testing the system before implementing it.

The system will have to accommodate for the "play-out" engine thread and the reporting of methods calls. This will yield some performance overhead in the system.

The XML file is parsed every time the application is started and will have to accompany the application. This is not desirable if intended to be used as a final implementation, and is a problem that would have to be resolved if the prototype was to be developed further. In the next chapter we will take a look at the possibility of generating code directly from a live sequence chart.

9 Direct Code Generation

9.1 Introduction

In this chapter we will focus on generating code directly from an LSC specification. LSC has the expressive power to create branching constructs and loops in addition to specifying message sequencing. It could be useful and practical to generate code directly from these constructs. The ability to generate code directly from an LSC model would circumvent the need to create the system behavior, but system structure in the form of class diagrams would have to exist in order to produce a working application. Figure 9.1 shows the implication of this on the development process described in chapter 6.

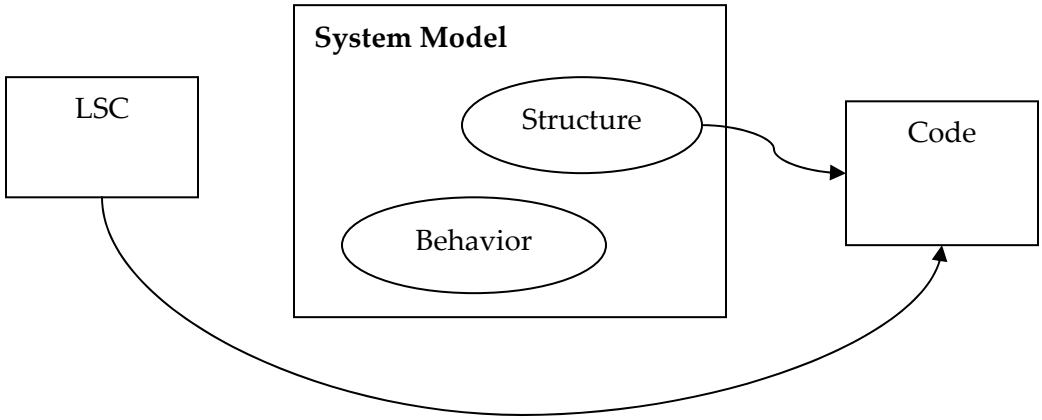

Figure 9.1: The implication of direct code generation on the development process

We have focused on generating Java code, where messages in the LSC model reflect method invocations. Generating sensible code that is easy to read and understand has also been a goal with this approach to code generation. Code generated by third party tools from statecharts does not yield code that could be used as basis code in further development. As there is no way of specifying the return of a message in LSC all methods generated from the LSC model will have *void* as return type. All generated methods will be public.

It is easy to describe LSCs where direct code generation would not be very practical or in fact possible. Figure 9.2 shows a simple chart where this is somewhat evident.

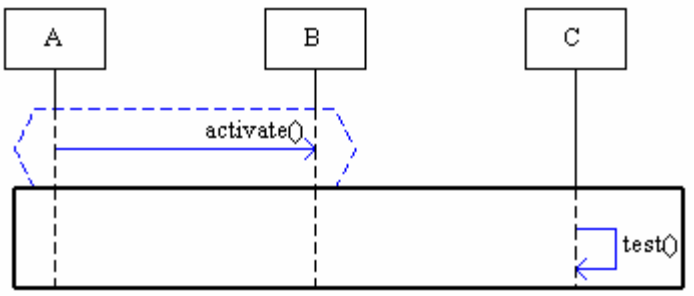

Figure 9.2: Simple live sequence chart

In Figure 9.2 the instance B has a method "activate()", when and only when an instance A calls this method should the universal part of the chart occur. The universal part of the chart describes that instance C must call the method "test()" on itself. Since there is only one message in the prechart, the main chart could be seen as the implementation of the method this message reflects, but the main chart itself does not make this trivial.

Instance B must know that it is called from instance A. There is no simple way of doing this without having a "process" that can know when A calls the method and when B receives the call and then report to B that it was in fact A that called it. There is no explicitly defined connection between instance B and the instance C where the method "test()" should be called. This means that B has no way of telling that C should call the method, and C has no way of knowing when it should call the method. In the "play-out" approach described in chapter 8 this problem is solved by having a "monitor" that all instances report to, and that has at its disposal an engine that can drive the execution in other instances. If our goal is to generate rational code, this approach is not convenient.

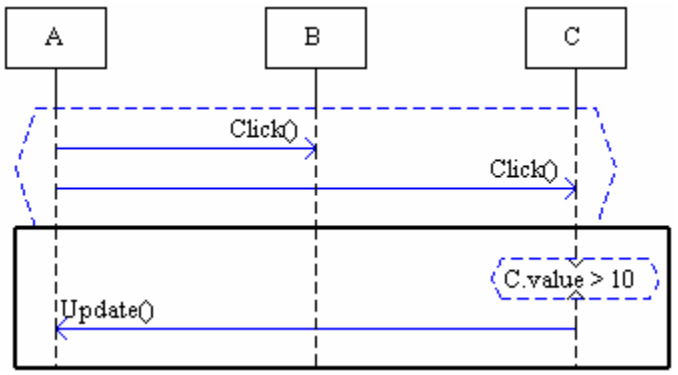

Figure 9.3: Two messages in a prechart

In Figure 9.3 it is hard to see the main charts as an implementation of any of the methods reflected by the messages in the prechart. First the method "Click()" has to be invoked on the instance B, then another method "Click()" has to be invoked on the instance C. A supervisor process would have to monitor the occurrence of the two methods and force

the system to exhibit the universal behavior when they occur. Again, this would not yield practical and easy to read code.

To allow for code to be generated directly from live sequence charts, the language of LSC has to be altered somewhat. We will focus on generating code on a chart by chart basis, and not take into account scenarios where several charts can be active at the same time and impose restrictions on each other. There can be no user interaction in the main charts, and when a main chart is executed it will execute until completed or violated by a "cold" or "hot" condition.

In the next chapter we will propose several constraints and simplifications on the LSC language. The goal of these alterations is to enable code generation on simple charts. Some alteration is done out of necessity, others are done for ease of generation.

9.2 LSC Language Constraints

A prechart can only contain one event that must be a message. The universal main chart will be seen as the implementation of the method the message in the prechart reflects. This means that the messages and events in the main chart must be described from the viewpoint of the instance that receives the message in the prechart.

An activation message could also be used to indicate for what method the main chart should be considered an implementation. We will focus on using a prechart as we later will use the XML format described in chapter 4 in a prototype for this approach. This format has no way of defining activation messages.

The instance names will only reflect the classes that are instantiated. This means that when a message is called on B in the prechart, the implementation of the method will be for the class B. A special instance name called "[Any]" is introduced. When this special instance is the sender of a message in the prechart it is of no importance which class the method is called from, and the implementation of the method can be seen from a global perspective.

Messages invoking methods in the main chart must be equipped with reference to the instance it is called on. This must be from the class that receives the message in the precharts viewpoint. If for instance a class B receives a message in the prechart, and in the main chart a message "test()" is to be sent from class B to class C. We will assume that class B has an attribute that instantiates an object of class C. We call this instance "myC". The message must then be written as "myC.test()" in the chart. This example can be seen in the Figure 9.4 below. The code generated from this chart is marked in bold character in Figure 9.5.

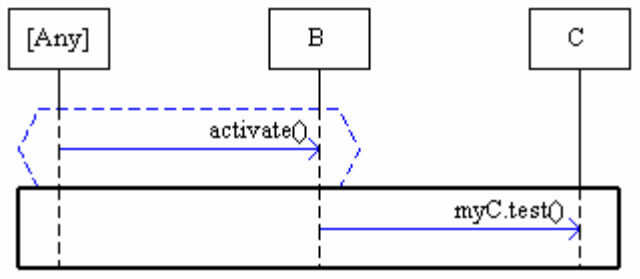

Figure 9.4: Example chart

```
Class B
{
    C myC = new C();

    void activate()
    {
        myC.test();
    }
}
```

Figure 9.5: Generated code in bold characters

All messages in the main chart must originate from the receiving class in the prechart. This is also the case for subcharts and conditions. All conditions must at least be synchronized for the receiving class, and the expression in the condition must be written from the viewpoint of the receiving class. Any expression that is valid based on the methods and attributes of the class is allowed. For subcharts at least the receiving class must be involved.

Subcharts can be used in collaboration with conditions to form loops and branching constructs. If charts are described using the simplifications presented here one can generate code for method invocation, branching constructs and loops. This is done by applying the rules described below.

9.3 Code Generating Rules

9.3.1 Conditions

When a "cold" condition in the main chart that is not associated with any subchart is violated the method is simply exited.

```
if(!(CONDITION_TEXT)) return;
```

If a “hot” condition is violated abort the system

```
if(!(CONDITION_TEXT)) System.exit(1);
```

9.3.2 Messages

If a message event is encountered, simply insert the message name. All messages in the main chart should be specified with full context, with instance name, operation and parameters.

```
MESSAGE_NAME;
```

9.3.3 Branching Constructs and Loops

Loops and branching construct of IF-THEN, IF-THEN-ELSE, DO-WHILE or WHILE types must be specifically identified.

If a single subchart without a loop definition is encountered and the first event is a “cold” condition, it should be considered an IF-THEN construct. Only messages and “hot” condition can be allowed in the subchart. Messages will be inserted and “hot” conditions will abort the system.

```
if(CONDITION_TEXT)
{
      // insert events in subchart, all of which must
      // be messages or “hot” conditions
}
```

If two subcharts with a transition between them are found, and the first event in the first subchart is a “cold” condition, it should be considered an IF-THEN-ELSE construct. Again, only messages and “hot” conditions are allowed in the subcharts.

```
if(CONDITION_TEXT)
{
  // insert message events from first subchart
  // insert “hot” conditions
} else {
  // insert message events from second subchart
  // insert “hot” conditions
}
```

There are basically two kinds of loops; finite and infinite. Finite loops are described in LSC with the number of iterations in the top left corner, for infinite loops an asterisk is used. Both finite and infinite loops can be of the form DO-WHILE and WHILE.

If the first event in the subchart is a "cold" condition it should be considered a WHILE loop. Other "cold" events in the chart can be accommodated for by breaking the loop if they are evaluated to be false. Message events and "hot" condition will be handled normally.

```
while(CONDITION_TEXT)
{
  // insert message events and "hot" conditions
  // for "cold" messages:
  if(!OTHER_CONDITION_TEXT) break;

}
```

For limited loops the code must keep track of the number of iterations. If several limited loops exist in each chart, they must be created with different iteration counters, or be created in a way that reuses the same counter.

```
int __count = 1;
while(CONDITION_TEXT && __count<=[LIMIT])
{
  // insert events
  __count ++;
}
```

If the last event in the subchart is a "cold" condition a DO-WHILE loop can be generated. If the loop is finite, the number of iterations has to be tracked in the same fashion as with WHILE loops.

```
int __count = 1;
do {
  // insert events
  __count ++;
} while(CONDITION_TEXT && __count<=[LIMIT])
```

A loop subchart where no specific first or last event is a "cold" condition, code can still be generated. A infinite WHILE(true) loop can be used. If the loop is limited it will loop until the count has reached the limit or a "cold" condition in the subchart is evaluated to be false. For unlimited loops false "cold" conditions will break the loops.

```
int count = 1;
while(__count<=[LIMIT])
{
  // insert message events
  // break on "cold" conditions
  // abort on "hot" conditions
  __count++;
}
```

When a subchart is not a loop and the first event is not a "cold" condition, code can be generated in the same fashion as with loops with no specified first or last "cold" condition. A DO-WHILE(false) construct can be used. This would at most be executed only one time, if a "cold" condition does not break the construct.

```
do
{
   // insert events
   // abort on "hot" conditions
   // break on "cold" conditions
} while(false)
```

9.3.4 Identifying Calling Class

If the calling class of the message in the prechart is not of the "[Any]" type, code can be generated to check what class called the method. It should be noted that in Java only the class that called the method can be obtained, not the instance. But since we have limited this approach for code generation to only consider classes this is not a problem. The code that would be inserted for an operation would look like this:

```
void operation(…)
{
      StackTraceElement[]trace = new Throwable().getStackTrace();
      if(!trace[1].getClassName.equals(CALLINGINSTANCE)) return;

      :
}
```

The ability to obtain the calling class of a method can allow for the possibility to accommodate for different behavior when the same method is invoked from different classes. If two charts have the same message in their prechart and the same message recipient, but different senders, both main chart implementations can be used in the methods code.

```
void operation(…)
{
      StackTraceElement[]trace = new Throwable().getStackTrace();
      if(trace[1].getClassName.equals(CALLINGCLASS_1))
      {
         // implementation of chart 1's universal part
      }

      if(trace[1].getClassName.equals(CALLINGCLASS_2))
      {
         // implementation of chart 2's universal part
      }
}
```

9.4 Prototype

We have developed a code generator prototype that can generate code from charts described with the limitations presented in this chapter. It uses an event-tree build from the events in an LSC XML file using the algorithm described in chapter 4. First the precharts are analyzed to see if it meets the qualifications presented earlier, if it does, the tree will be traversed starting with the first element node in the main chart. If code has been generated successfully from an element node it will be marked as traced, and before an element will be processed all its parent nodes will have to be marked as traced. This is done assure that the nodes are processed in accordance with the partial order of elements.

The rules presented above only allow for generation of methods. Class diagrams from a system model structure can coupled with this approach to generate complete classes with methods. This is done in our prototype, and allows the methods generated from the LSCs to be merged with a Java system model described by XMI. First code is generated for the methods in the LSCs, and then a modified version of a XMI code generator we developed for an earlier project [20] is used to generate code for classes and methods. When the generator encounters a method in the XMI structure it will look for this method among the LSC generated methods. If a method is found in the LSCs, this is used in the class, if not, the XMI method is used.

9.4.1 Television Example

We modify the charts from the Television example to comply with the simplified LSC specification. The system model will also have to be extended somewhat from what was presented in chapter 8.

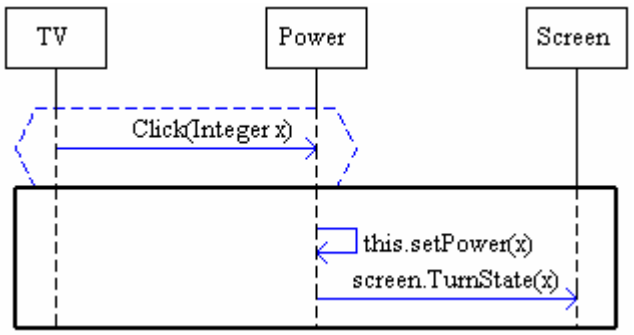

Figure 9.6: Turning on the television

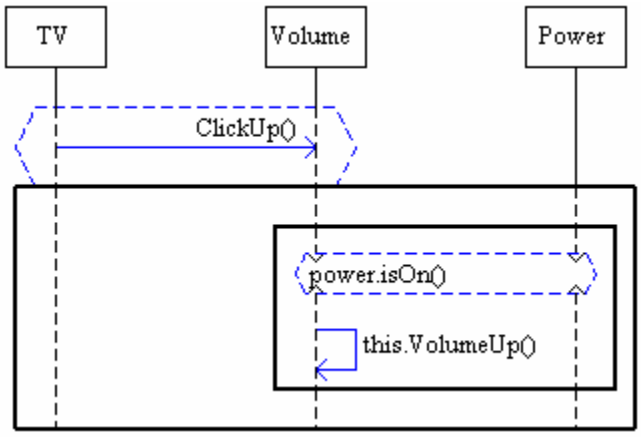

Figure 9.7: Lowering the volume

The system model for the Television example has been refined in a more implemental direction. The classes for Volume, Screen and Channel have been equipped with references to the power control. The Power class has been extended with a method for sensing the power state, and with a reference to the screen.

After generating code from the charts and merging this with the system model, the television becomes operational. The power can be turned on and off, volume can be raised and lowered and the channel can be changed.

In Appendix A instructions for using the direct code generation prototype can be found.

9.5 Summary

The methods described in this chapter impose a severe limitation on the language of LSC, but allows for the possibility of generating code directly from requirements in the form of LSCs. The generated code is limited but might prove useful in certain cases. For simple applications it can circumvent the need for designing a system model. The ability to merge the generated code with an existing system model renders the creation of complete working applications from partial implementations and their requirements possible.

In the next chapter we will discuss the technologies introduced in this paper. We will also discuss our work and the results of the prototypes.

10 Discussion

10.1 Introduction

This thesis starts with a presentation of the language of live sequence charts and its elements and constructs. We continued by presenting a proposed methodology for executing live sequence chart models, called "play-out". We isolated the most fundamental elements of the language and constructed a XML language for describing basic charts consisting of them. An algorithm for taking events listed in the XML format and building a tree structure that describes the partial order of these events was then developed. Last, three approaches of code generation for LSCs was introduced and placed in the context of a software development process and then in separate chapters solution sketches for the three approaches were presented.

This discussion will have the same structure as this thesis in general. The following elements are discussed.

- The language of LSC and its extensions compared to classical sequence diagrams.
- The methodology of "play-out" and which applications this technology might have.
- Our own work on the creation of the XML format and the event-tree algorithm.
- The three code generation possibilities, their different range of application and our prototype implementation of two of them.

10.2 The language of live sequence charts

With LSC being an extension of MSC one would expect their expressive power to supersede that of MSC. LSC does in fact do this. With the possibility to define possible and mandatory behavior they are already in a league of their own. The presented extensions on the LSC language further enhance its expressive power.

While still a relatively new technology it is not yet adopted in many facets of the software industry, but as tools supporting LSCs are developed this will surely change. We have been informed by D. Harel that a book with the title "Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine" will be released shortly. This book will be accompanied by the "play-engine" software for use by the reader.

LSC maintains the appealing visual power of MSC while improving its expressive weaknesses. These improvements make it possible to seriously look at ways of executing requirements and synthesizing inter-object behavior models like statecharts.

10.3 Playing out behavior

"Play-out" as described previously is a proposed method for executing live sequence charts directly. The idea is that the underlying functionality of the system should not have to be implemented, but rather the live sequence charts should be used to describe it. Parts of the system that can not be described by LSCs and system structure would have to be implemented. The "play-out" methodology will interact with the system and give the illusion of operating the system as it was a final implementation.

One of the more obvious uses for "play-out" of a system would be as a stage of the development phase. The user would be able to describe how he wanted the system to operate and the developer could then make a simple model, containing a user-interface only coupled with live sequence charts. They could together interact with the system and examine if the requirements were as the user specified and if the system operates satisfactory. The developer could rectify whatever the user was not satisfied with before implementing the system.

We have in chapter 8 argued that for some reactive systems the execution of the behavioral requirements based on "play-out" could suffice as a final implementation. While this may hold true, we can only envision this to hold for smaller and simpler systems. In large and complex system the amount of charts might prove difficult to handle for the "play-out" methodology and dead-lock situations and violation of charts might occur. The execution of complex system would in most cases require the aid of formal analysis, which has been described as "smart play-out".

"Play-out" of live sequence charts is an excellent tool for testing applications. With the constraints set in the language it allows one to specify behavior of the system explicitly, there is nothing left to randomness if modeled correctly. When executing these charts any inconsistency of the system will be detected, if one such inconsistency exists. The existential charts make for good possibilities to specify specific scenarios and when incorporated into the model and executed you get the liveness of mandatory and possible behavior.

10.4 XML format for LSC and partial order trees

The LSC language along with all of its additional extensions yields quite a comprehensive expansion of message sequence charts. To accommodate for all these elements in an XML format would be a formidable task. When developing our format we focused on encompassing the basic and most often deployed LSC elements. One of our goals was to create a format that could be edited manually with little effort, because of this there are certain limitations in the format.

The limitations in our XML format includes only being able to have one event for each location as well as not supporting time-enriched charts. While this format proved to be

sufficient for our needs, it is not the most robust format. Not only could support for additional LSC features be included, even some of the existing elements can be enhanced. The XML elements for conditions and assignments could be more "parameterized" which would allow for easier processing by tools.

Using XMI for storing LSCs would be another option. We have been informed that a utility that does this is under development by Omega [22], but was not available during our work on the thesis.

The algorithm for building event-trees that describe the partial order of events was created for use with the XML format we constructed and for our specific use in the code generation aspect of the thesis. While we found it to be versatile and useful, it is unclear if it is powerful enough should the code generation approaches be developed further. The algorithm is lacking in some ways, and does not distinguish between the sending and reception of message events. A message is considered a single event, and asynchronous messages are therefore not supported. This produces a more manageable tree, and since we were dealing with code where a method invocation can be considered a synchronous message, the simplification was acceptable.

The “LSC Visualizer” we created for the presentation of live sequence charts is not fully developed, but it shows the power of the XML format combined with the event-tree. The application is very sensitive to errors in the format, and it has not been rigorously tested.

10.5 Code generation

In this thesis we have studied three different applications of code generation, each with different area of use.

10.5.1 Synthesizing Statecharts from LSC specification

The use of synthesis to create statecharts or an approximation thereof will be the approach that has the most value in a typical development process. The generated statecharts can be refined until adequate behavior is captured, and then well developed and properly tested third-party tools can be deployed to generate the final code.

For statecharts to be synthesized from an LSC specification, the charts need to go through severe testing algorithms to prove consistency and satisfiability of the specification. While this testing may yield a global system automaton that should in theory be able to be converted to statecharts, the algorithm for this is at the time of writing not yet satisfactory developed. Converting basic charts with only synchronous messages is possible, but there has been no effort to our knowledge at this point to develop methods for including synchronized conditions over several instances, branching constructs and loops. The methodology presented in [4] focus mainly on high

level mathematical proofs and algorithms, and not processes that are easy to adapt in tools. There seems to be some way to go there still before this becomes a viable option for code generation.

The code generated from a system model with statecharts tends to become complicated and further editing of this code without comprehensive knowledge about statecharts would be almost impossible. Unless this results in a full implementation, the value of the code can be disputed. The only tool available to us during the work on this thesis was Poseidon for UML with a code generator from statecharts beta plug-in, unfortunately the code generated would not compile.

10.5.2 Generating code directly from LSC specifications

The direct code generation methodology we have proposed is quite limited in several aspects. The most important limitation is that all of the charts would have to be designed specifically for the use in direct code generation. The interaction of several LSC charts that imposes restraints upon each other and describe different facets of the same scenario will not be possible. One chart will be considered the implementation of the one method referred to in the prechart. This apart, the approach might be useful in some scenarios. For instance when developing small applications and want to circumvent the need to develop system model behavior.

For a simple program such as our Television application, we showed that it was possible to go from an application with no behavior to a fully operational one, although the LSC charts and the partially implemented system model were changed specifically to allow this.

10.5.3 Integrating “Play-out” with a partial implementation

Generating code into an existing implementation in order to utilize a “play-out” engine to execute LSC diagrams describing system behavior can be very useful for testing partial implemented systems. This approach could also accommodate for the full expressive power of the LSC language, as is not the case with synthesis or direct code generation which is based on a restricted subset of LSCs. The “play-out” methodology is the technology that is most matured in comparison with direct code generation and synthesis of statecharts. A robust implementation of it is incorporated in the “play-engine” software presented in [2].

Another great advantage this approach has is that it could yield a final implementation. A skeletal implementation for the application would already exist. For instance a user interface where one can press the buttons, but nothing happens, or database interaction methods. The LSCs would fill inn the blanks, and the result would be going from an “empty” implementation to a “full” implementation, an application that actually works.

For this to be achievable for large systems, we would need a well developed and robust "play-out" engine with formal verification "smart play-out" algorithms implemented.

When developing our prototype for this approach we found that it produced some performance overhead. A thread for the "play-out" engine was needed as well as a monitor receiving report of every method in the LSC charts.

10.5.4 Comparison

We have in this chapter discussed three different approaches of code generation from live sequence charts. These methods differ greatly. The integration of "play-out" uses code generation as glue between the LSCs and the system model. This yields an implementation of the system without implementing it. Converting the LSCs to statecharts would probably yield the most complete implementation. Direct code generation can hardly be seen as viable once the systems become larger and more complex. In which of these three different approaches the futures lies is a question that remains to be answered. Direct code generation will fall further behind the two other technologies once these have been streamlined. If "play-out" or statecharts yields the best system would probably be on a case to case basis.

10.6 Further work

The monitor can be altered to generate live sequence chart from the "developed" software, and these charts can be compared against those that were made as the behavioral requirements of the software before they were implemented. This could allow for a more rigorous testing of the behavior requirements with the help of algorithms presented in [4] for consistency testing.

The algorithms for synthesis of statecharts from an LSC specification leave much to be desired at the moment. They could be developed further to take into consideration all of the elements and constructs of live sequence charts. The algorithm presented in [4] should be considered a draft. Tools with the methodology implemented must be developed before the power and usability of it can be proven.

The prototype for the integration of a "play-out" engine into an existing system still has some way to go before it can be used with the full expressiveness of the LSC language. For now only basic charts are possible, with conditions, subcharts and message events. But with the implementation of the remaining LSC elements and "smart play-out" it could be a great tool for testing systems or as an engine driving a final implementation.

11 Conclusion

In this thesis we have studied an introduced extension of message sequence charts called live sequence charts, which improves expressive weaknesses in message sequence charts and introduces the notion of mandatory behavior in charts. We have also studied a methodology for executing charts described by LSCs, called “play-out”.

Tools that support LSCs were not available when writing this thesis, so we had to develop our own algorithm and format to be able to process charts in our own tools. We created an XML format for describing live sequence charts that should both be easy to create manually and be useful for computerized tools. We also constructed an algorithm for building a tree structure that describes the partial order of events within a chart. These two elements were combined in a tool for visualizing LSCs called “LSC Visualizer”. The “LSC Visualizer” will parse XML files and draw the charts and events trees described.

We continued by giving an introduction of a development process and then presented three different approaches of code generation that was placed in context of this development process.

The method of using synthesis to generate the system model behavior directly from the LSC specification and from there use conventional tools for generating code from the system model is the approach that has most to give in the development process. It “fits” with the holistic development process and introduces an automated step in going from behavioral requirements to system model. As we concluded in chapter 7, this methodology still has some way to go before applicable in the software development process. We see a lack of step by step procedures for conversion that are adaptable by tools. This solution is also dependant on good “third-party” software for generating the actual code from the statecharts.

The “play-out” methodology is proposed method of executing behavioral requirements described by LSCs. We have described a solution sketch for coupling a partial implementation with a “play-out” engine and created prototype components for such a solution. The main area of application for this method is for testing the requirements of the application. Detecting errors in the requirements and correcting them before an implementation of the system has been done would be of great value in complex systems and can reduce the development costs greatly. A “play-out” method that uses formal verification called “smart play-out” is under development and the use of this in a “play-out” engine would increase the quality of execution. We have argued that if the executed behavior is sufficient in capturing the intended behavior of the system, this approach could for some systems be used for a final implementation. The implementation would have to accommodate for the performance overhead the “play-out” engine produces.

We have also explored the possibilities of generating code directly from LSCs. While some constraints on the LSC language was proposed in order to do so, this approach might still have its uses for small applications, or as an instructive way of exploring what an LSC specification actually means. The prototype we have developed for this generates code for methods gathered from LSC charts. These are in turn merged into an existing system model, and thus creating a working application.

Live sequence charts are expressively powerful while still maintaining a highly visual and intuitive way of constructing behavioral requirements. They are far more expressive than MSC and are therefore more suitable for specifying actual behavioral properties of reactive systems. Our work has shown that they have a wide range of applicability in the development process, although there might remain some work before they reach their full potential.

As commented in section 10.6 the algorithm for synthesis of statecharts from live sequence charts must be improved before the possibility outlined in chapter 7 is viable. Further work must also be done on our "play-out" engine before it could accommodate for the full expressive power of LSC and be a really useful tool for developers.

Abbreviations

CD – Compact Disc
GSA – Global System Automaton
GUI – Graphical User Interface
ITU – International Telecommunication Union
JDK – Java Development Kit
LSC – Live Sequence Charts
MSC – Message Sequence Charts
SDK – Software Development Kit
TV – Television
UML – Unified Modeling Language
XML – Extensible Markup Language
XMI – XML Metadata Interchange

References

[1] W.Damm and D.Harel. LSCs: Breathing Life into Message Sequence Charts. Available from: http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/LSCs.pdf [Accessed January, 2003]

[2] D.Harel and R.Marelly. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. Available from: http://www.wisdom.weizmann.ac.il/~dharel/papers/JSM03.pdf [Accessed January, 2003]

[3] D.Harel and R.Marelly. Playing with Time: On the Specification and Execution of Time-Enriched LSCs. Available from: http://www.wisdom.weizmann.ac.il/~dharel/papers/TimedLSCs.pdf [Accessed January 2003]

[4] D.Harel and H.Kugler. Synthesizing State-Based Object Systems from LSC Specifications. Available from: http://www.wisdom.weizmann.ac.il/~dharel/papers/synthesis.ps [Accessed January, 2003]

[5] R.Marelly, D. Harel and H.Kugler. Multiple Instances and Symbolic Variables in Executable Sequence Charts. Available from: http://www.wisdom.weizmann.ac.il/~dharel/papers/OOPSLA02.pdf [Accessed January, 2003]

[6] D.Harel. Can Behavioral Requirements be Executed? (And why would we want to do so). Available from: http://www.wisdom.weizmann.ac.il/~dharel/papers/CanReqsBeExecuted.ps [Accessed January, 2003]

[7] D.Harel, H.Kugler, R.Marelly and A.Pnueli. Smart Play-Out of Behavioral Requirements. Available from: http://www.wisdom.weizmann.ac.il/~dharel/papers/FMCAD02.pdf [Accessed January, 2003]

[8] D.Harel. From Play-In Scenarios to Code: An Achievable Dream. IEEE Computer, 34(/1):53-60, January 2001. Available from: http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/PlayInToCode.pdf [Accessed January, 2003]

[9] Object Management Group (OMG), www.omg.com [Accessed May 23, 2003]

[10] Gentleware, www.gentleware.com [Accessed May 23, 2003]

[11] Rational Software, www.rational.com [Accessed May 23, 2003]

[12] I-Logicx, www.ilogix.com [Accessed May 23, 2003]

[13] Z.120 ITU-TS Recommendation: Message Sequence Charts (MSC). ITU-TS, Geneva, 1996.

[14] International Telecommunication Union (ITU), www.itu.int/home/ [Accessed May 23, 2003]

[15] Documentation of the Unified Modeling Language (UML), available from the Object Management Group (OMG). Available from: http://www.omg.org/technology/documents/formal/uml.htm [Accessed May 23, 2003]

[16] Extensible Markup Language (XML), 1.0 (Second Edition), W3C Recommendation October 2000. Available from: http://www.w3.org/TR/REC-xml [Accessed May 23, 2003]

[17] XML Metadata Interchange (XMI), http://www.omg.org/technology/documents/formal/xmi.htm [Accessed May 23, 2003]

[18] Microsoft framework SDK http://msdn.microsoft.com/downloads/ [Accessed May 23, 2003]

[19] J. Klose and H. Wittke, An Automata Based Interpretation of Live Sequence Charts. T. Margaria and W. Yi (Eds.): TACAS 2001, LNCS 2031, pp. 512-527, 2001.

[20] T. Homme, J.E. Ramsland, O. Stallemo and K. Vatne, Kodegenerering fra XMI (Codegeneration from XMI), Available from: http://fag.grm.hia.no/ikt2340/projects/ThomasHomme/Rapport.pdf [Accessed May 23, 2003]

[21] Java Development Kit (JDK), Available from: http://java.sun.com [Accessed May 23, 2003]

[22] Omega. http://www-omega.imag.fr [Accessed May 23, 2003]

[23] D.Harel "Statecharts: A Visual Formalism for Complex Systems". Available from: http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf [Accessed May 21, 2003]

[24] Harel and UML Statechart comparison: Available from: http://asusrl.eas.asu.edu/est/State/Statecharts%C2%AD_Lian.ppt [Accessed May 17, 2003]

[25] B.P. Douglass, UML Statecharts Introduction. Available from: http://www.embedded.com/1999/9901/9901feat1.htm [Accessed May 23, 2003]

Appendix A – Using the prototypes

General Information

All of the code generation tools are written in Java with JDK 1.4.1 [21], backwards compatibility has not been tested. The prototypes have been deployed as .jar files for convenience; these files include precompiled .class files for the prototypes. The relevant directory structure can be seen in Figure A1.

```
CDROM
  |
  |
  \---- DirectCG
  |     |
  |     \---- source code
  |
  \---- LSC Visualizer
  |     |
  |     \---- source code
  |
  \---- Message Reporting
  |     |
  |     \---- source code
  |     |
  |     \---- TVExample
  |
  \---- PlayOut
  |     |
  |     \---- source code
  |
  \---- TVExample
```

Figure A1: Directory Structure

Note: All examples given assume that the folders are copied from the CD to a folder on the hard drive. This is because the prototype will generate output when compiling the TV example and place these in a subfolder. Both examples assume that the Java runtime environment "jre" is in your PATH or java.exe and javac.exe is given with its full pathname.

Using the Direct Code Generation Prototype

On the CD accompanying this thesis under the folder \DirectCG\ an XML file for the LSCs of the Television example DC_TVExample.XML is found, together with a version of the television system model DC_TVExample.XMI. To generate code from this, copy the DirectCG folder to your hard drive, enter that folder and type the following commands:

```
Java –jar Generate.jar DC_TVExample.XMI DC_TVExample.XML
```

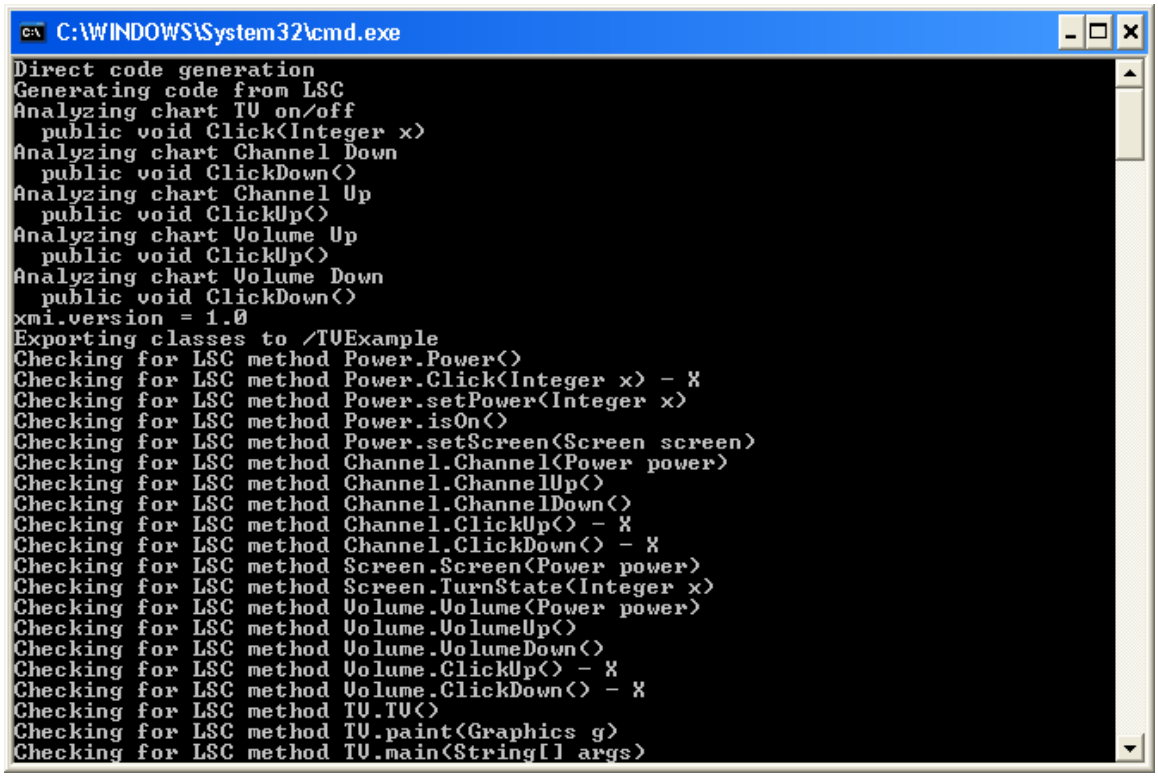

Figure A2 : Generating the Television example

Figure A2 shows the output when code is generated for the Television example with direct code generation. First all the charts in the LSC specification are analyzed and methods are generated from them. Next the generator checks all the methods encountered in the XMI file against the methods generated from the LSC specification. If they match the method is replaced by the LSC method. This can be seen in the screenshot for the where the checked operation is marked with an "X".

The code will be generated into the folder \DirectCG\TVExample. The Java files TV.Java, Power.Java, Screen.Java, Volume.Java and Channel.Java will be located there. In the folder \DirectCG\TVExample type these commands to be able to compile and run the TV application.

```
Javac *.java
Java TV
```

The source code for direct code generation prototype can be found under the folder \DirectCG\Source Code\.

Using the "Play-out" Prototype

First the message reporting must be adapted into the system model. Copy the "Message Reporting" folder to some place on the hard drive. In the folder \Message Reproting\ run the command:

```
Java -jar Generate.jar TV.XMI TVExample.XML
```

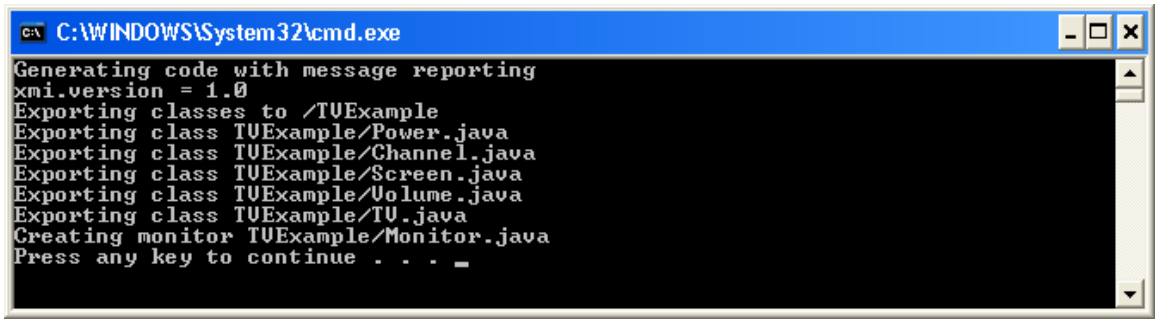

Figure A3 : Generating code with message reporting from an XMI system model

Code for the system model with message reporting integrated will be generated into the folder \Message Reporting\TVExample. The Generated files will be TV.Java, Power.Java, Screen.Java, Volume.Java, Channel.Java and Monitor.Java. These files need a "play-out" engine and the LSC specification in XML format to be able to run. The files PlayOut.jar and TVExample.XML exist in the folder \Message Reporting\TVExample for this purpose. To run the TV application with "play-out" driving the universal part of the LSC specification type commands below in the folder \Message Reporting\TVExample. Note that semicolon in the class path specification is followed by a punctuation.

```
Javac -classpath PlayOut.jar;. *.java
Java -cp PlayOut.jar;. TV
```

Source code for the generating message reporting can be found under \Message Reporting\Source Code\ and the code for the "play-out" engine can be found under \Playout\Source Code.